Design and implementation of a massively multi-player online historical role-playing game

David Alexander Bond
H00124719

February 2015

Computer Science
School of Mathematical and Computer Sciences

Dissertation submitted as part of the requirements for the award of the degree of MSc in Advanced Internet Applications

DECLARATION

I, David Alexander Bond, confirm that this work submitted for assessment is my own and is expressed in my own words. Any uses made within it of the works of other authors in any form (for example: ideas, equations, figures, text, tables, programs) are properly acknowledged at any point of their use. A list of the references employed is included.

Signed: DABond

Date: 13-02-2015

ABSTRACT

The goal of this project was to design and evaluate the initial version of a distributed, scalable game engine, the *JominiEngine*. The project focused on developing the essential core of the *JominiEngine*, and used it to instantiate *Overlord: Age of Magna Carta*, a game set on mainland Britain in the time period of 1194-1225. Future projects will extend, by degrees, the functionality of the *JominiEngine*, ultimately resulting in the production of a fully featured, adaptable MMORPG (Massively Multiplayer Online Role-Playing Game) platform.

Wargames and historical simulations have been popular pastimes since the last half of the 20th Century, and their value as tools for the teaching and reinforcement of creative decision-making is increasingly being applied across a whole spectrum of occupations. The arrival of computer gaming has resulted in a huge increase in popularity for this type of game; in particular, with the advent of the Internet, the last two decades have seen a huge global increase in the player-base of MMORPGs.

The development of a historical MMORPG, therefore, presents the developer with challenges in the areas of both game design, in which a balance has to be reached between the requirements of historical accuracy and player enjoyment, and system architecture, which poses significant challenges with regard to system complexity and scalability.

The aims and objectives of the project were to develop: i) A core ***game model***, specifying the basic game objects, interactions, rules, mechanisms, and victory metrics. ii) A core ***game engine***, enabling players to perform the functionality outlined in the game model. iii) A ***system architecture***, enabling the efficient operation of the game engine at varying usage levels.

Particular attention was paid to the key areas of: ***historical accuracy***, to ensure educational value and player immersion; ***modularity***, to allow for the future expansion of the *JominiEngine* in subsequent projects; ***scalability***, to ensure that the game supports up to several thousand concurrent players, at an acceptable level of performance.

A game model was developed that stipulated basic class types (PC, NPC, Fief, and Army), key game roles (king, herald, system administrator), and specified component mechanics, rules and victory conditions. The design process was both thorough and systematic, and UML-style diagrams were used extensively, providing a well-documented basis for subsequent work.

Using this game model, a core game engine (the *JominiEngine*) was developed and implemented that allowed players to use the specified mechanics to manipulate objects within the game world in the manner defined by the rules. Modularity was facilitated through the provision of thorough documentation and clear interfaces to game and system components, and existing protocols were used where possible. Historically accurate data was imported into the *JominiEngine* to allow *Overlord* to be fully instantiated.

The choice of a non-relational (NoSQL), distributed DBMS, *Riak*, as the backend database for the system architecture is relatively novel, and should help to address the issue of scalability through its ability to sustain performance during periods of high usage, and its distributed nature, which could reduce latency for a widely distributed player base. Additionally, *Riak*'s distributed architecture also provides a built-in redundancy in the event that one of its nodes should fail.

Suggestions were made for future developments, including various expansions to the *JominiEngine* game model, full implementation of client-server architecture, the addition of game content authoring (modding), and the introduction of a full graphical user interface.

ACKNOWLEDGEMENTS

I'd like to thank the following people for their help with this project:

- Hans-Wolfgang Loidl for his support and wise counsel throughout and for somehow managing to keep my spirits up. Thanks also to Sandy Louchart for some very useful feedback and suggestions on the game design phase.

- My family and friends for their unceasing support and for not sounding too bored when I talked about nothing else except this project for eight months solid.

- My very understanding and flexible employers for allowing me to take this course, giving me the time to attend lectures and finding a way to fit my 'work' work (you know, the stuff I'm actually paid for) around my studies. Also my work colleagues for quietly listening to me drone on about my lost weekends.

I'd like to dedicate this dissertation to my wife, Ann, and to my late father, Peter Bond, both of whom I know would have been cheering me on from the side-lines.

CONTENTS

1. INTRODUCTION

Wargames and historical simulations have been popular pastimes since the last half of the Twentieth Century. Furthermore, their value as tools for the teaching and reinforcement of creative decision-making has been recognised since Georg von Reisswitz first introduced his wargaming rules in the early 19th Century. Indeed, this aspect is increasingly being applied not only in the military profession but also for managerial personnel across a whole spectrum of occupations.

The arrival of computer gaming has led to an increased flexibility not only in game design but also in the way in which games are played, and this has resulted in a significant increase in popularity. In particular, with the advent of the Internet, the last two decades have seen a huge global increase in the player-base of Massively Multiplayer Online Role-Playing Games (MMORPGs) such as *EVE* and *World of Warcraft*.

The development of a historical MMORPG, therefore, presents the developer with challenges in two main areas:

- Firstly, game design, in which a balance has to be reached between the requirement for historical accuracy and the need to keep the game enjoyable for its players.
- Secondly, system architecture which poses significant challenges with regard to system complexity and scalability, requiring a well-documented modular design with the application of clear interfaces.

1.1 Aims and objectives

The goal of this project is to design and evaluate the initial version of a distributed, scalable game engine, the *JominiEngine*, for a historically-based MMORPG set in a medieval context, entitled *Overlord: Age of Magna Carta* (hereafter referred to as *Overlord*).

It is intended that this be the first in a series of projects that will extend, by degrees, the functionality of the *JominiEngine*, ultimately resulting in the production of a fully featured, adaptable MMORPG platform allowing the creation of games in which players will assume the role of nobles and will manage fiefs, generate income, raise armies, fight wars with other players, and ensure their own family's lineage.

This project will focus on developing the essential core of the *JominiEngine*, and use it to instantiate *Overlord*, a game set on mainland Britain in the time period of 1194-1225.

1.1.1 Aims

The aims for this project are to develop the following components:

- ***A core game model***, specifying the basic game objects, interactions, rules, mechanisms, and victory metrics (see *Figure 5* in *Section 3.1 Methodology* and *Figure 9* in *Section 5.1 Game model design*). It is intended that the complexity of the initial version be reduced by scaling down the number of game components; therefore, the model is to be designed in such a way as to facilitate future development (see *Section 1.3: Requirements and specification*, for more details).

- ***A core game engine***, enabling players to perform the functionality outlined in the game model (see *Figures 10, 11* and *12* in *Section 5.2 Game engine design*). Only a minimal user interface is to be provided, sufficient for testing purposes, although it is hoped to provide a simple hexagon map of the game world for added context.

- ***An underlying system architecture***, enabling the efficient operation of the game engine at varying usage levels.

1.1.2 Primary objectives

Primary objectives for this project are:

- The creation of an ***interactive core game engine***. In order to achieve this, it is necessary to selectively reduce game scope and complexity, whilst still maintaining useful core functionality, and facilitating future expansion.

- ***Modularity***: it is important to regard the project in the overall context of an ongoing development process. To ensure for future expansion, therefore, the game engine must be designed with clear, well documented interfaces and protocols to all game components. Furthermore, interfaces should be of a high enough level to allow the replacement of underlying components (for example, the system database) without the need for extensive re-engineering.

- ***Scalability***: ensuring that the game will support up to several thousand concurrent players, at an acceptable level of performance. This requires the appropriate selection of system components and load balancing mechanisms. It is intended to assess scalability through the measurement of latency (i.e. how long it takes the game engine to process commands).

1.1.3 Secondary objectives

Secondary (optional) objectives for this project are:

- Content authoring: providing a facility for players to modify game content. This can be achieved through the exposure of game data, which can then be ported into the game by the player with the use of a scripting language.

- Availability: providing a sufficient level of redundancy to allow for the failure of key system architecture components.

1.2 Background

Unsurprisingly, given the increasing commercial significance of computer gaming, particularly multiplayer gaming, both the fields of game design and game system architecture have seen a growth in the amount of research being undertaken.

1.2.1 Game design

Research into game design covers a broad range of subjects, including:

- Game analysis. This includes: the morphological or ontological classification of games in order to facilitate both further study and game development; The identification of the key mechanics that should be present in a game and the ways in which these mechanics can be accessed by the player; Investigations into the motivation of players as a way of identifying aspects of games that may attract or repel potential game consumers.

- Technical aspects of game design. This area includes:
    - How to ensure modularity and increase game engine efficiency through the use of different programming paradigms, such as Object-Oriented Programming (OOP), Data-Oriented Design (DOD) and Entity Component System (ECS).
    - The way in which player interest can be maintained through use of scripting or content authoring systems that allow the modification and extension game content.

- Of particular interest to this project is the use of historical accuracy and how it can increase the immersive aspects of gaming, whilst also enabling the game to become an educational tool. This extends the relevance of the project beyond the immediate domain of computer science, to also include the social sciences.

1.2.2 System architecture

The area of game system architecture has seen research into a variety of approaches, including:

- How to efficiently balance the game load. Proposed solutions include distributing players according to interaction, dividing the game world into distinct but connected zones, using P2P architecture to dynamically balance the load amongst players, and the modularisation of functions in order to reduce the load on the core engine.

- How to minimise network traffic. Proposed solutions include basing communication around the player's AOI, judicious use of the publish-subscribe pattern, and the reviewing of standard networking practices such as obtaining object locks and the use of acknowledgement messages.

- How game data is stored. Proposed solutions include the use of relational or non-relational DBMSs, the use of distributed and non-distributed DBMSs, and the use of P2P architecture to share data storage amongst players.

With both game design and game system architecture in mind, this project focuses on:

- Firstly, the design and implementation of a ***core game engine***, initially of reduced functionality, that attempts to combine playability with historical accuracy.
- Secondly, an investigation into the suitability of a ***distributed NoSQL database*** to store game data, and to address the issue of scalability.
- Thirdly, being mindful of future projects, it attempts to ensure ***modularity*** and to expose game data in order to facilitate the future introduction of content authoring.

1.3 Requirements and specification

1.3.1 Functional requirements

1.3.1.1 Game model

As planned and described in the Research Report, in order to focus on technical aspects of the system architecture, the game functionality for this project contains core components only, including a reduction in the number of object attributes and game mechanisms.

It was decided that the game model should contain the following basic classes, along with any additional classes that may be required in order to allow them to perform their functions:

- PlayerCharacter (PC), controlled by the player. Players should be able to manage fiefs (raise income), raise and lead armies, move through the game world, interact with NPCs and with other players (for example exchanging money and troops), and conduct family affairs (to ensure family lineage).

- NonPlayerCharacter (NPC), controlled by a very simple AI. The NPC should be able to manage fiefs, lead armies, and move through the game world.

- Fief. A fief generates population and income, quantities of which will depend on fief attributes (for example, fields, industry, tax rate) and the attributes of the fief manager (player or NPC).

- Army. An army is a collection of troops of different types; it can move through the game world and can be used in a defensive or aggressive role against enemy armies or fiefs. The effectiveness of an army will depend on a variety of factors, primarily numbers and strengths of troops, and the attributes of the army leader (player or NPC).

In addition to the above, the game model should provide the following roles:

- Faction leaders (i.e. kings). This role acts as the focus for a faction and has increased income and powers within the game (for example, appointing nobles to honorary positions).

- Faction heralds. These are essentially helpers and facilitators, providing advice to faction members, helping the king with his finances, and generally enabling the smooth running of the game.

- System administrator (sysadmin). The sysadmin performs an overview role for the entire game system and have access to all administrative functions, including the editing of game objects. The sysadmin is strictly neutral and is available to provide help and advice for any players.

Where possible, within the limitations of game and the level of abstraction, game mechanisms and data should be historically accurate; *Overlord*, the concrete instance of the game should reflect the period of 1194-1225.

The game model should provide channels of communication in order to facilitate the formation of alliances between PCs, and to allow other game-related communication. This could be achieved mainly through external tools, such as third-party chat systems, rather than deeply integrated into the game engine.

1.3.1.2 Game engine

It was decided that the game engine should:

- Provide a minimal user interface, through a test client, allowing the player to:
  - Interact with game objects, as defined by the game rules.
  - View their current holdings, finances, and armies.
  - View their current location within the game world.
  - View their in-game progress and ranking.
  - (Optional) View their current location and holding (and those of their faction) on a hexagon map.

- Allow players to communicate with each other through chat and/or bulletin boards.

- Allow players to register accounts, enabling them to participate in games; also, allow players to delete their accounts, removing any account information.

- Allow players to see games in progress and to join them where there are places available.

- (Optional) Provide a facility for players to modify existing game content or introduce new content; alternatively, design the architecture in such a way as to facilitate the future addition of this functionality.

1.3.1.3 System architecture

It was decided that the underlying system architecture should:

- Provide a means for storing both persistent game world data, data associated with individual games, and data metrics. This should be achieved through the use of a database management system (DBMS); it should allow for large numbers of small database transactions, system scalability, and the sophisticated interrogation of data metrics.

1.3.2 Extra-functional requirements

1.3.2.1 Game engine

It was decided that the game engine should:

- Be designed to run on the following operating systems: i) Microsoft Windows (version 7 onwards); ii) Any Linux-based operating system, allowing it to be hosted on a machine in the Linux labs of the School of Mathematical & Computer Sciences, Heriot-Watt University

- (Optional) Allow for the future development of a graphical user interface (GUI).

1.3.2.2 System architecture

It was decided that the underlying system architecture should:

- Allow scalability up to thousands of concurrent players; this will be tested through the measurement of system latency.

- In the interests of modularity, provide clear interfaces to all game and system components and a clear definition of protocols used to exchange data. In order to facilitate this, where possible, standard protocols and specifications will be employed; for example, the GEDCOM specification could be used to communicate and display PC genealogical data.

- (Optional) provide sufficient redundancy to ensure availability in the event of the failure of key system architecture components.

1.4 Report outline

Section 2 (Literature review) provides an overview of the research that has been undertaken in the fields of game design and game system architecture, highlighting those areas that are key to this project. A comprehensive list of references is provided in a separate section at the end of the report.

Section 3 (Methodology) covers the decisions taken with regard to the game engine design (both strategic design concerns and specific implementation issues), and reviews the environment, software tools and languages used in the course of the project, giving the reasons for their choice.

Section 4 (Project management) addresses issues concerned with the management of the project, including proposed project plan (timeline); risk assessment; professional, legal, ethical and social issues; project management methodology.

Section 5 (Game design) describes the creation of the game model, explaining any decisions taken regarding the core game mechanics and rules. It also describes how the model was translated into the initial design of the game engine, detailing the proposed structure and key features.

Section 6 (Implementation) describes how the game design was implemented and the manner in which the project was conducted, giving details and explanations of any game design or project management decisions that were taken in the process.

Section 7 (Reflection) analyses the degree to which the project succeeded in accomplishing its aims and objectives, highlighting both the main achievements and any known gaps and limitations, and drawing general conclusions. It reflects upon the performance and suitability of the technologies used, draws attention to any immediate tasks needing to be undertaken, and makes suggestions and recommendations for medium- and long-term extensions to the game model and system.

Section 8 (Conclusions) succinctly highlights the main outcomes of the project, and provides suggestions for the ways in which future work may build upon those outcomes.

Please note that supplementary information and data, such as test tables and formulas used in the game engine, can be found in the appendices at the end of the report.

2. LITERATURE REVIEW

Although, as stated in *Section 3: Objectives*, the main focus of the project will be concerned with technical aspects of the underlying system architecture (for example, how to ensure scalability), it is necessary to look at a number of other areas, some of which will form part of future projects (see *Section 4: Future developments*). These include game design, content authoring, and historical background.

2.1 Game design

Anderson et al. (2010) survey the area of 'serious games', defining them as "computer games that are not limited to the aim of providing entertainment, that allow for collaborative use ... for learning and educational purposes" (Anderson et al., 2010, p.255). The review focusses on the use of 'serious games' in the field of cultural heritage, a genre which has in recent years seen a growth in popularity with the release of commercially successful games such as the *Civilization* and *Total War* series.

An increasing number of commercial history-based games have become available, many based on historical conflicts. These games often contain a high degree of historical accuracy and range from 'traditional' wargames that depict specific battles in which units are abstractly represented on hexagon maps, to those that use cutting edge animation technology to allow the player to actually take part in battles.

As Kirschenbaum (2011) points out the value of wargames as tools for the teaching and reinforcement of creative decision-making has been recognised since Georg von Reisswitz first introduced his wargaming rules in the early 19th Century. Indeed, this aspect is increasingly being applied not only in the military profession but also for managerial personnel across a whole spectrum of occupations. He asserts that the wargame is "a vehicle for its participants, either through role-playing or the arbitrary rule-based constraints of the game world, to critically examine their own assumptions and decision-making processes." (Kirschenbaum, 2011)

Video games are seen as an ideal medium for attracting the interest of young people, and the availability of high performance video gaming engines has led to the development of games specifically targeted at cultural heritage education - Anderson et al. mention the *Virtual Egyptian Temple*, for example, developed by Jacobson and Holden (2005) and the *Ancient Olympic Games*, developed by Gaitatzes et al. (2004).

Anderson et al. also draw attention to the increasing provision of modding tools as an intrinsic part of many commercial games has allowed some of them to be adapted for educational purposes; for example, *The History Game Canada* (unfortunately no longer available) which adapted *Civilization III* in order to model Canadian history.

The use of AI to model realistic behaviour in the denizens of virtual worlds, and thus create a more immersive experience, has become also increasingly important, say Anderson et al. Programming techniques are being continuously developed to enable game entities to implement decision-making in order to achieve in-game goals and objectives. These techniques include the use of Finite State Machines (FSMs) which defines a set of distinct behavioural states for an entity, only one of which can be active at any one time, and which are triggered by in-game events. Anderson et al. also mention Goal-Oriented Action Planning (GOAP) which relies on the entity dynamically decides upon a plan of action based on the goal, available actions, and the costs associated with those actions. Another technique being developed to further enable AI is that of annotated environment, in which the objects that make up the game world contain the necessary knowledge (data) that allow other game objects to interact with them. This allows game entities to 'learn' new behaviour simply by adding data to the environment, removing the need to continually re-write their functionality.

2.1.1 Game design: Game analysis

An essential first step in the creation of a computer game (or, indeed, any computer system) is to identify the component parts, objectives, player motivations, etc. This information can then be used to develop a template for the proposed system.

Sicart (2008) attempts to define the concept of game mechanics in the context of object-oriented programming, allowing the inter-relationship of the mechanics to be more clearly mapped, and also showing how they relate to game hardware and player experience.

He refers to research carried out by Avedon (1971) into the structural components of games, which makes the distinction between game rules (permitted actions) and mechanics (how they are carried out). Sicart sees game mechanics as "actual interaction" and rules as the "possibility space" in which some mechanics can be accessed.

He asserts that a more useful definition for the game programmer is that "… game mechanics are methods invoked by agents, designed for interaction with the game state." This definition allows mechanics to be described in an object-oriented way, which is more useful to the description of the game as a software system and can be more easily described using existing object-oriented tools such as UML diagrams. Additionally, this definition is not limited to the requirement for human interaction but can be applied to automated entities - i.e. an agent could be a PC or an AI controlled NPC.

Sicart is of the opinion that games should use several categories of game mechanics, some of which relate to the principal aims and challenges of the game, and others that allow the player to perform actions that are not necessarily aimed at achieving those central goals. The game should allow for the player having their own goals, and should provide players with the freedom to pursue them. This 'sandbox' aspect of games is becoming increasingly important, particularly in MMORPGs, and is seen as a way of encouraging player's to invest their time and energy in the game.

Sicart defines core game mechanics as those that are repeatedly used to achieve the victory game-state, as defined by the game designer. He further breaks core mechanics down into two sub-types - primary and secondary. Primary core mechanics are those that can be directly used to overcome problems, resulting in the achievement of the victory game-state. Secondary core mechanics are not essential to achieving the victory state and may only be available under certain circumstances, but they may nevertheless aid the player in achieving victory. This distinction allows designers and programmers to apply a finer granularity to the mechanics they provide to their players.

Sicart also defines concept of 'compound mechanics' as a sub-set of game mechanics that comprise a core mechanic, although it is often useful to think of them as single mechanic for purposes of initial game design.

Sicart feels that this definition of game mechanics can be used in relation not only to game entities such as PCs and NPCs but also to other game components such as input devices. Additionally, it lends itself readily to transcription into documentation and thus aids communication between designers and programmers.

Crawford (2013) argues that there is a significant gulf between those who design games, whose outlook tends to be artistic and creative, and those who program them, who tend to approach things in a more procedural and disciplined manner. As a result, the final version of a game will often not reflect the designer's vision.

In order to bridge this gulf, Crawford (2002) encourages game designers to think of their games in terms of verbs - what actions will the player want or need to be able to perform in the game?

> "At the outset of the design process, after you have established the goals of the design but before you have begun work on the design itself, you must ask yourself the question, "What are the verbs in this design?" All through the design process, you must ask yourself that question time and time again." (Crawford, 2002, p.93)

Crawford (2013) states that, when thinking of game objects, the designer should consider them in terms of what actions they perform. "A window is not glass; it's something that blocks air movement while permitting light to pass freely." (Crawford, 2013, p.90) Such thinking can help highlight basic actions and inter-object relationships, which can then be mapped to methods in the programming stages.

Crawford (2013) also uses the concept of 'personality models' in order to define the way in which game characters (PC and NPC) interact with each other. To do this he proposes assigning each character a number of variables, divided into several types: intrinsic, mood, volatility, accordances, and relationships. Intrinsic variables are the basic attributes, such as intelligence, integrity and attractiveness. Mood variables cover the extent to which a character exhibits basic emotions such as anger and sadness. Volatility variables control the rate at which a character's moods change (for example, adrenaline effects anger). Accordance and relationship variables dictate the way in which a character will perceive the intrinsic variables of other characters.

Crawford (1984) also urges game designers to closely consider the relationships between opponents which can define the whole tone of the game. Is this relationship asymmetric, in which each opponent is assigned a unique combination of resources and attributes, or symmetric, in which opponents receive exactly the same resources and victory just depends upon execution? The former often makes for a more interesting game but tends to be harder to program, as goals need to be carefully balanced in order to be achievable by both sides.

Designers should, Crawford asserts, attempt to introduce the concept of triangularity into their games; this is the ability to pursue victory through the use of indirect strategies, which may also leave the player open to defeat. This concept can be extended to allow victory to be achieved through third parties, such as NPCs or even PCs. Such aspects of gaming can, Crawford feels, help to reveal certain intrinsic truths about society: "… society discourages direct conflicts. Yet conflict remains in our lives, taking more subtle and indirect forms … Only indirect games offer any possibility of designing games that successfully explore the human condition." (Crawford, 1984, p.72)

Lankosk and Björk (2008) propose a novel approach to game design by basing the game structure around the design of its characters and their interaction. Their proposed method is based around three main aspects of character design: i) Recognition: how a character is defined in the game world, both physically and emotionally; ii) Alignment: how the character is controlled and what kind of access is given to their thoughts and motivations. A way of defining alignment, for example, is by associating with a particular social group; iii) Allegiance: the characteristics of a character that allow a player to positively identify with them. This aspect can be very important in regard to a player's immersion in the game: As Lankosk and Björk point out, "Successful allegiance makes players feel that what they are doing in the game is the right course of action since they buy into the goals of the PCs." (Lankosk and Björk, 2008, p.4). This is an especially important aspect for role-playing, where giving a player moral choices that can be rewarded or punished both increases their engagement with the game-world and reinforces their sense of freedom.

Alverez et al. (2006) argue that the classification of computer games is intrinsically difficult because of the constant evolution of computer technology. They attempt to overcome this problem by creating a tool - *V.E.Ga.S.* (*Video Entertainment & Games Studies*) – that takes a morphological approach to classification; that is, based on a game's component parts and how they are structured.

Using a seed collection of 588 games, several areas were analysed including: interactivity (how the player interacts with the game world); actions; rules; results. The last three areas can be easily mapped to what programmers might refer to as a function - i.e. it takes an input (action), defines how it can be transformed (rules), and this produces an output (result).

Alverez et al. used the data from this analysis to derive a number of 'game bricks', a combination of which can be used to classify any game. The following twelve bricks were identified: answer; manage; have luck; shoot; construction/creation; block; destroy; move; avoid; position; time; score.

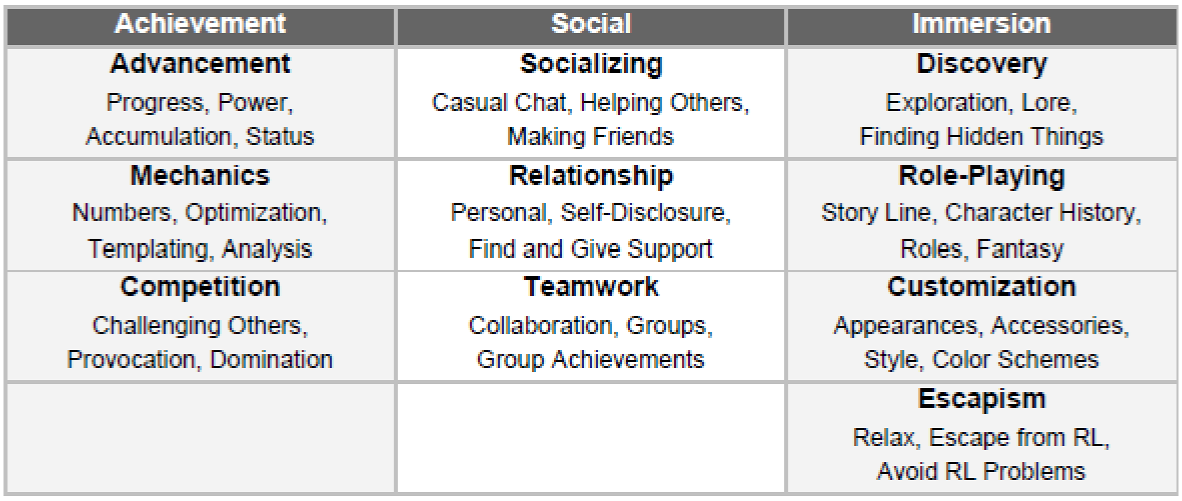
Alverez et al. found that it was possible, using the game bricks, to identify a number of game 'families' (games that had the same combination of bricks) and that, furthermore, these families often contained games from different genres. In this way, it was possible to reveal underlying themes that were not easy to see at first glance.

Alverez et al. also identified a number of 'metabricks' (bricks that always accompany each other). For example, 'shoot' always accompanies 'destroy' and can be combined into the metabrick 'killer'. Using this method the following metabricks were identified: driver; killer; god; brain. These metabricks allow games that were classified in unique families (a family containing only a single game) to be reclassified as belonging to other broad families.

Yee (2006) attempts to create an empirical model of the motivations of MMORPG players, arguing that the ability to classify player motivations, in combination with usage or subscription figures, could be a powerful tool for not only game designers and publishers, but also for researchers (for example, in the field of psychology).

Early research in this area was carried out by Bartle (1996), who identified four main Player Types – killers, socialisers, achievers, explorers – and attempted to define the ways in which they interacted with each other and with the game world. Yee points out, however, that the assumptions Bartle made to arrive at his model have never been empirically tested.

Yee compiled a questionnaire containing forty questions (each with a rated answer of 1-5), based on Bartle's player types, with additional input from previous MMORPG surveys. Data was then collected from 3,000 participants. Ten main motivational categories were derived from the results, each of which contained a number of sub-categories. For example, the 'advancement' category contains 'progress', 'power', 'accumulation' and 'status'. These categories were further sorted into three main groupings – 'achievement', 'social' and 'immersion'.



Achievement	Social	Immersion
Advancement Progress, Power, Accumulation, Status	**Socializing** Casual Chat, Helping Others, Making Friends	**Discovery** Exploration, Lore, Finding Hidden Things
Mechanics Numbers, Optimization, Templating, Analysis	**Relationship** Personal, Self-Disclosure, Find and Give Support	**Role-Playing** Story Line, Character History, Roles, Fantasy
Competition Challenging Others, Provocation, Domination	**Teamwork** Collaboration, Groups, Group Achievements	**Customization** Appearances, Accessories, Style, Color Schemes
		Escapism Relax, Escape from RL, Avoid RL Problems

Figure 1: MMORPG player motivations (Yee, 2006)

The results revealed some interesting correlations. For example, whilst both males and females are equally sociable, females tend to be more interested in relationships; Males are more competitive than females; Analysis seemed to indicate that the 'escapism' motivation could be correlated to problematic (i.e. unhealthy) usage.

Yee was also able to disprove one of Bartle's assumptions, namely that Player Types are independent of each other; Yee's results showed that players with a strong preference for one motivational category can, in fact, rate highly in others.

Aarseth et al. (2003) attempt to define a system for more accurately identifying genres of 'virtual world' games by using to set of 'dimensions' (i.e. game aspects), grouped under five main space, time, player structure, control, and rules (see *Table 1*).

Dimension	Explanation
Space	
Perspective	Omni-present (the player can see everything) or vagrant (can only see what the character sees)
Topography	Geometrical (continuous freedom of movement) or topological (discrete non-overlapping moves – e.g. chess)
Environment	Dynamic (the environment can be changed by player or game events) or static (non-changing - e.g. a football field)
Time	
Pace	Real-time or turn-based
Representation	Mimetic (events realistically mimic real-world event time) ors arbitrary (events occur at set pace)
Teleology	Finite (there is a clear ending point for the game) or infinite (the game can just keep going - e.g. The Sims)
Player structure	
Player structure	Single player, multiplayer, single team, etc.
Control	
Mutability (of character)	Static (e.g. the player is just awarded points), experience-levelling (the character's stats are permanently enhanced), or power-ups (the character

	receives a temporary boost to powers)
Savability	Non-saving, conditional (e.g. at certain ‘checkpoints’) or unlimited
Determinism	Deterministic (the same outcome can be reproduced by carrying out the same actions) or non-deterministic (there is an element of randomness)
Rules	
Topological-rules	Victory rests on the presence of the player at a certain place in the game world
Time-based-rules	The passing of time effects victory status
Objective-based-rules	Victory is based upon specific conditions being met

Table 1: Aarseth’s ‘dimensions’ for game classification

It is Aarseth et al.’s goal that the classification of games using the above method may lead to the design of new games by combining dimensions in new ways; taking an existing game and altering some dimensions could radically alter its nature in ways that had not initially occurred to the designer.

2.1.2 Game design: Scripting and content authoring

Anderson (2011) reviews the field of scripting systems deigned to modify and expand game content, and attempts to classify existing systems.

Although originally only used by game developers as a tool in the game design process, the inclusion of scripting systems, be they generic or proprietary, provide the player with the opportunity for content authoring (also known as ‘modding’), providing the potential to significantly expand the game’s scope. Indeed, Anderson argues that “scripting systems are considered one of the most important developer tools that are included in modern game engines.” (Anderson, 2011, p.47). Such systems also allow the development of ‘serious games’ - educational historical simulations – through the adaptation of existing commercial game technology, and has also enabled the commercialisation and licencing of game engines such as *Unreal*.

Many games now include fully-featured integrated development environments (IDEs) for the purpose of content modification (or ‘modding’) which has led to the increasing popularity of the data-driven programming paradigm. Data-driven design separates system behaviour from system functionality and architecture, allowing behaviour to be defined by input data.

This has the effect of freeing game designers and artists from the constraints of working with program code and, correspondingly, allows programmers to progress more quickly without having to wait for designers. The result is a transformation from internal coded program logic to external game asset, states Anderson, reducing the complexity of the core game engine. Anderson defines a scripting system in this context as “a system using a programming language which allows the modification of program logic without the need to recompile the application (game engine) source code.” (Anderson, 2011, p.49).

Anderson’s review finds that scripting systems are often interpretive rather than compiled, which increases ease of understanding (an important issue for end-users who may not be trained programmers) but reduces performance. However, many are designed to be pre-compiled and executed within a virtual machine (VM) which makes for an improved performance and a reduction of run-time errors. Many scripting systems are embeddable within the core game engine, and trend is towards generic rather than proprietary languages.

Anderson's classification defines 3 main categories of scripting system, based on functionality: i) Initialisation systems, which are run once, usually to set the initial values of parameters, etc. at game startup. ii) Trigger-only induced systems, which either define actions to be taken by game engine when certain event occurs, or define both the event and subsequent actions. Iii) Traditional program-like systems, which contain familiar programming structures (for example, control loops).

Anderson finds that many languages can be included in all of the categories above. One such language is *Lua*, which may account for its increasing popularity. Binstock (2013) asserts that scripting languages "have already become standard practice in game development, where *Lua* is frequently embedded for scripting UI components."

White et al. (2009) also recognise the potential benefit to be gained from using scripting languages which "allow developers to easily specify how an object or character is supposed to behave, without having to worry about game itself" (White et al., 2009, p.43); they argue that it is of special importance for MMOs because of the sheer number of objects and players present in game. The use of scripting languages has sometimes resulted in the development of completely new games (for example, *Counter-Strike* from the *Half-Life* source), although it is often necessary to limit their capabilities in order to reduce the occurrence of software errors and cheating.

White et al. also draw attention to some problems, however. A drawback to many traditional scripting languages (e.g. *Python*) is that the user must be aware of some low-level issues (such as frame rates) that are crucial to game performance but which most amateurs and even designers are often not be familiar with. There have also been problems, particularly in MMOs, arising from running many scripts concurrently, resulting in inconsistent game state. Many of these problems are being solved by adapting scripting languages to make use of tried and tested OOP patterns, such as the state-effect pattern which ensures that an object's state can be updated by many concurrent effects (i.e. the results of actions that can change the state) in a more structures way.

Another problem pointed out by White et al. arises from the way in which scripts can interact with each other in subtle and unforeseen ways; this means that an incorrect data value in one script can cause an error in another. To try to help identify where these problems might occur, work is being carried out in the area of data provenance (i.e. tracing where data originates, how it has been altered, how it is transferred between entities, etc.) but this is proving difficult to introduce into scripting languages.

2.1.3 Game design: Programming paradigm

Another approach to game design, advocated by West (2007), is the use of Entity Component System (ECS) programming. West asserts that the traditional object-oriented approach to games programming often results in three common problems: Firstly, leaf objects can become over-burdened by the presence of superfluous functionality further up the inheritance tree. Secondly, some functions are only available to very specific types of objects; if the programmer wants to allow another type of object to invoke it, they either have to duplicate the functionality or move functionality further up the inheritance tree. Thirdly, over time, as the code is modified, objects tend to morph into blob anti-patterns.

With ECS, on the other hand, game objects are replaced by an entity composed of a series of separate components. Rather than being a hard-coded object, each entity is defined by its constituent components and each component contains the methods that are used in its processing, independent of other components. Components should be as independent as possible but, pragmatically, there may need to be dependencies between certain components (for example, component A may need to be processed before component B); such dependencies need to be addressed through programming.

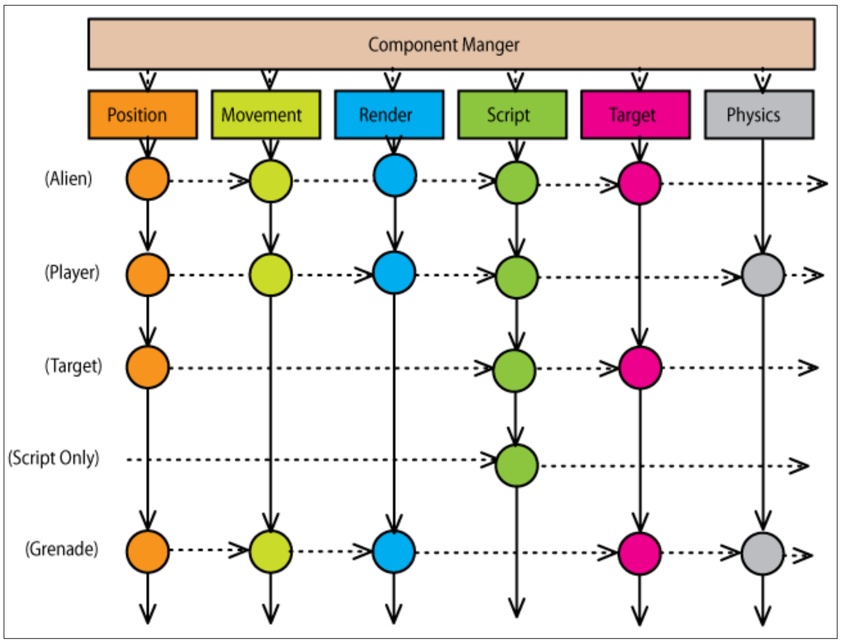


Figure 2: Object composition using components, viewed as a grid (West, 2007)

Such an approach lends itself to data-driven programming and enables the simple creation of new objects from existing components. In turn, this provides designers with the freedom to experiment without need for significant coding.

Martin (2007) argues that the use of ECS for MMORPGs allows for increased efficiency in processing large numbers of game objects based on common characteristics (i.e. components), a feature that is increasingly useful as MMORPGs grow in both game world size and graphical sophistication. Also, when used in combination with an underlying database, it allows objects to be defined fluidly using queries, rather than a coded list of components.

Another programming paradigm to consider when designing games is Data-Oriented design (DOD), advocated by Llopis (2009). Llopis argues that, with its dependency on grouping code around a perceived object, "object-oriented programming (OOP) and the culture that surrounds it … could be hindering your project rather than helping it."

Conversely, DOD shifts the focus of the program to the data itself and how it is handled. DOD specifies that a program should be structured in a way that is most suitable for the processing of the data; where possible data should be grouped by type, even if it is from different objects. In this way, the programmer can address a single aspect of multiple objects simultaneously (see *Figure 3*).

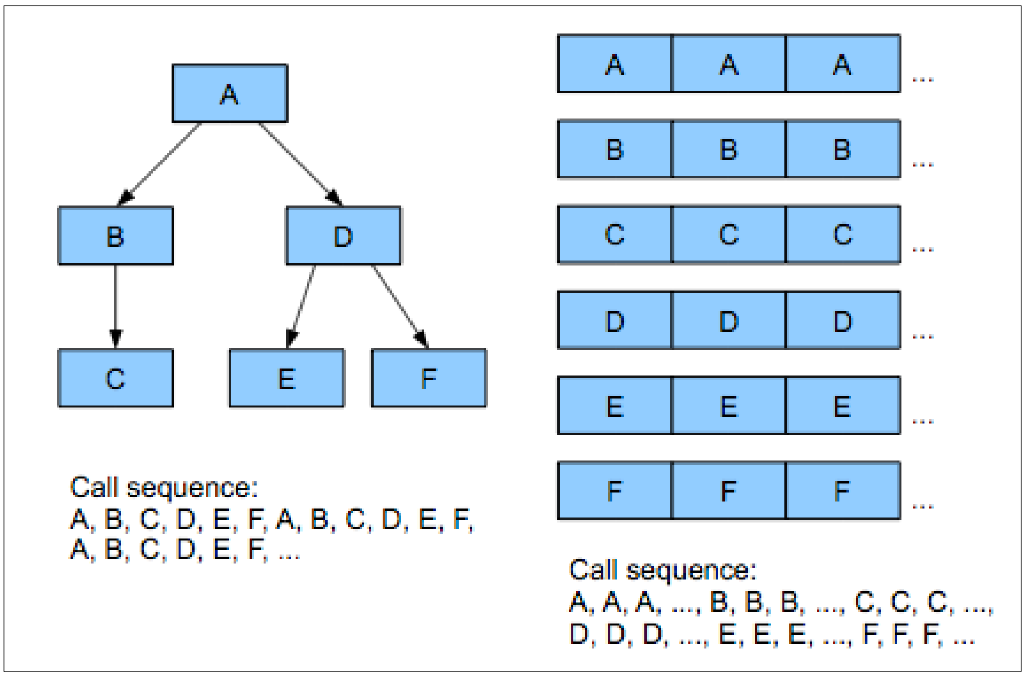

A
B
D
C
E
F
Call sequence:
A, B, C, D, E, F, A, B, C, D, E, F,
A, B, C, D, E, F, ...
A A A ...
B B B ...
C C C ...
D D D ...
E E E ...
F F F ...
Call sequence:
A, A, A, ..., B, B, B, ..., C, C, C, ...,
D, D, D, ..., E, E, E, ..., F, F, F, ...

Figure 3: Data calls in object-oriented (left) and data-oriented (right) programming (Llopis, 2009)

Llopis claims that DOD can help with a number of issues: Firstly, parallelisation because it enables the processing of separate data types in different threads or processors with a minimal need for synchronisation. Secondly, cache usage because the same code is being used consecutively rather than different consecutive functions for the various data types in an object. Thirdly, modularity because functions designed to work on single data types are often small and simple to follow. Finally, testing because tests only need to supply an input and check on the output.

However, Llopis also draws attention to some disadvantages: Firstly, there can be a significant overhead when learning DOD, as most developers will have been taught object-oriented programming. Secondly, it can be very difficult to interface a DOD program into an existing object-oriented one. Finally, DOD can make debugging harder because data relating to a single logical entity can be stored in different structures and be processed separately.

2.1.4 Game design: Historical background

Dunnigan (2000) argues that a key feature in the design of wargames and historical simulations is accuracy. “The object ... is to enable the player to recreate a specific event … [therefore] the game must be realistic” (Dunnigan, 2000, p.13). This places an extra burden on the game designer who needs to ensure accuracy, both in game data and mechanisms, regarding aspects such as geography, society, and military technology and doctrine.

However, Dunnigan points out, it is also important to remember that it is a game and should therefore be fun to play; most players of wargames and historical simulations are looking for what Dunnigan terms as ‘dynamic potential’ – how the player might interact with the game world and mechanisms in order to change history.

It is important, therefore, that the game model should be based around issue of conflict and the attainment of victory conditions that make sense in this context. Essentially, the game should be concerned with how best to manage resources in order to achieve this aim. The in-game battles will be abstracted but should reflect general concepts within the historical context (for example, how military technology and practices affected the relative strengths of troops for each nationality).

Two very useful sources for wargaming, emerged from the aftermath of the Napoleonic wars: Von Clausewitz (1997) provides much useful information on the principles and conduct of warfare, covering topics such as the foundations of strategy, the importance of information and planning, and the best uses of offense and defence. Of more practical use to game designers, von Reiswitz (1824) produced what is regarded as the first modern set of wargaming rules, aimed specifically at the military profession and since translated and adapted by military establishments in many countries around the world. It defines a rigid set of rules and charts that can be used to simulate the events and effects of battle.

The location and period selected to be the subject of the MMORPG in this project is Western Europe in the high and late medieval periods (roughly AD 1100 to 1450); conflicts of this period include the First Baron's War, the Hundred Years War and the Wars of the Roses, all of which are the subject of many historical works.

The pioneering work by influential historian Charles Oman (1885) chronicles the development of military practices throughout the medieval period, including the increasing importance of siege warfare. Although somewhat venerable, the work of Oman is significant because of its ability to construct a coherent narrative from fragmentary and often partisan sources. Although primarily a study of the 'military art', Oman's work always attempts to place this topic within a political context: "… to speak of the characteristics of military science involves the mention of many political institutions … Feudalism, in its origin and development, had a military as well as a social side, and its decline is by no means unaffected by military considerations." (Oman, 1885, p.1)

Tuchman (1979), whose work is concerned with the Hundred Years War, describes the late medieval period as "a violent, tormented, bewildered, suffering and disintegrating age" (Tuchman, 1979, p.xv) that historians have come to see echoed in the tumultuous events of the Twentieth Century, particularly those surrounding the two world wars. Tuchman argues that the very fact that we can see, from a vantage point of 600 years in the future, certain similarities in the behaviours of people and societies, shows them to be "permanent in human nature" (Tuchman, 1979, p.xvi) and therefore valuable in the lessons that they can teach.

However, Tuchman points out that it is very hard to chronicle events of this period for a number of reasons; for example, dates were calculated differently (the year began at Easter, which varied from year to year), populations and numbers in armies were almost always exaggerated, different sources spelt names in different ways, and sources generally contradict each other.

Tuchman's work is very much a narrative history of the Hundred Years War, grounded by following the life of one particular participant, the French noble Engeurrand de Coucy VII. However, it is nevertheless a valuable source of data on the late medieval period; for example, giving a succinct explanation of the names, values, and weights of the coins and currency of the age.

Sumption's 3-volume work (1990, 1999, 2009) on the Hundred Years War attempts to not only chronicle the events of this epic conflict but also to place them within the context of medieval society, showing the ways in which it shaped that society. For example, he recounts how the delicate socio-economic balance that existed between the towns and the rural areas was often disrupted by the influx of farmers who, due to privations caused by conflict, abandoned their small holdings for a new life in the town.

Like Tuchman, Sumption also draws attention to the unreliability of medieval sources, finding most chronicles to be "episodic, prejudiced, inaccurate and late" (Sumption, 1990, p.x), often because they were written under the patronage (and therefore reflecting the interests) of particular aristocrats. Other sources, however, such as civil and military records, frequently contain much unbiased and illuminating information.

Nofi and Dunnigan (1997) provide a wealth of information not only about the Hundred Years War but also about medieval life and society in general, including warfare, customs, economy, and life expectancy. This illustrated online book was compiled as a companion to Nofi and Dunnigan's MMORPG, *The Hundred Years War*, (no longer available) and contains much information of direct relevance to the prospective MMORPG designer, whilst also attempting to put the events of that conflict into the context of happenings in the wider world.

Gillingham (1990) gives an account of the Wars of the Roses, the series of conflicts that arose in England shortly after the conclusion of the Hundred Years War. He attempts to divorce the events of these wars from the myths that have since come to dominate our perception of them, thanks in no small part to the works of William Shakespeare. Gillingham provides some very useful information about medieval life in England and helps to explain why the nature of that conflict was so very different from that of the Hundred Years War. For example, because of an extended period of peace in England (in total contrast to continental Europe) there were very few towns with walled defences and, therefore, very few sieges in the Wars of the Roses. This led to short, sharp campaigns in which the opposing sides tried, in general, to bring each other to battle as soon as they could gain an advantage.

Nicole (1996) provides a compendium of facts relating to military science and practices in Europe throughout entire medieval period (AD 400 to 1400), sub-divided into distinct periods, one of which is the late medieval period (AD 1275 to 1400). He includes data on recruitment, organisation, siege warfare, taxation, logistics and strategy.

There are also a number of sources written specifically for designers of MMORPGs; these aim to provide game world data for games that either take place in the medieval era or in a fantasy setting, most of which are firmly based on medieval society and demographics. Steele (2012) provides a detailed and thorough study of the fief, very much the key building block of medieval society, concentrating on life in England and France during the middle to late medieval periods. Topics covered include fief ownership and responsibilities, agriculture, economy, taxes, population, warfare and society in general.

Written as a companion piece to the work above, Steele (2010) concentrates on the entity of the European town and city during the late medieval period, giving a detailed overview of all aspects of life, including commerce, taxes, laws, governance and urban society. Being mindful of her intended audience (game designers), Steele attempts in both of these works to supply many useful facts and figures (for example, military wages and prices for common trade items).

Similarly, Ross (2013) provides concise, usable data that focusses particularly on late medieval Europe, aiming to help instil a sense of realism to the game world. Topics covered include population density (cities and rural), economy and agriculture, and fortifications.

Other, more narrowly focussed, texts can also prove useful when trying to introduce realism and authenticity:

- Broadberry et al. (2011) focus on the English population during the high to late medieval period and attempt to arrive at accurate estimates by considering sources such as the Domesday Book, poll tax returns, and manor records; these population estimates were also considered in the light of agricultural records, to see if they could be supported by the amount of food available.
- Adams (2004) examines the way in which Lanchester's Laws can be used to arrive at a reasonably realistic estimate of battle casualties, given the numbers involved and the rough historical period. Lanchester's Linear Law can be applied to 'ancient' warfare (i.e. where battle was essentially a man-to-man affair) and predicts that, given similar weapons and troop quality, casualties will be roughly equal on both sides. Lanchester's Square Law is used for modern combat and takes into account the application of ranged weapons, giving the larger army an even more significant advantage than would at first be apparent.

2.2 System architecture

Caltagirone et al. (2002) identify a number of key goals that an MMORPG must fulfil. They include:

- The provision of a game world that contains customisable objects (thus providing the role-playing element)
- A game world state that is persistent (i.e. that exists independently of a player's gaming sessions)
- The ability for players to interact both with other players and with the game world

To this list Driel et al. (2011) also adds:

- A game world state that is not only persistent but also *consistent*, between the client and the game servers, and also between the clients themselves

In order to provide these components, the MMORPG architecture must face various issues (Caltagirone et al., 2002), including:

- The minimisation of network traffic
- Load balancing
- Efficient client-side performance
- Scalability
- Security

2.2.1 System architecture: Models

Caltagirone et al. identify two main models of architecture that are commonly used in the design of the MMORPG – centralised, or client-server (C-S), in which the game state and logic are stored in and processed by servers before being communicated to the client to allow viewing and interaction by the player; Peer-to-peer (P2P) in which the game state and logic are stored in and processed collaboratively by the clients. Driel et al. (2011) adds a third type, namely the hybrid architecture, which contains elements of both C-S and P2P.

Caltagirone et al. give several reasons why C-S is traditionally seen as the model of choice for MMORPGs:

- Unlike P2P, it allows the game to be controlled centrally, which is a more acceptable business model for many game publishers.
- Being hosted on dedicated servers which are available 365 days a year, it is easier to maintain game world persistency and consistency without the need for the complex synchronisation required by P2P.
- It is more easily secured against cheating, as each client request can be verified against the centrally-stored game state before it is actioned.
- It is a simpler architecture to understand and implement.
- Through the proper application of scalability, any potential server bottlenecks, a disadvantage often associated with the C-S model, particularly during periods of high usage, can be mitigated.

Proponents of P2P, often make the claim that the level of network traffic is lower, thus reducing latency, but this is a much-debated issue. Caltagirone et al. cite Cronin et al. (2001) as claiming that C-S and P2P have roughly equivalent network traffic profiles, whilst Diot and Gautier (1999) claim that P2P only generated half the network traffic of C-S.

2.2.2 System architecture: Client-Server distributed functions

Modularisation of functions, in order to provide for the most efficient running of the core game engine, is a strategy suggested in a number of articles. Dieckmann (2013) proposes a multi-server architecture with the following separate components:

- Underlying the whole system, a dedicated database; this ensures persistence and provides a definitive source of data in case of system failure.
- A login server; this allows for the most efficient processing of logins and, being a distinct function, is ideal to separate. In addition, to protect the player's personal information, enhanced security measures can be more easily implemented.
- A patch server; this enables the updating of clients prior to login, allowing a potentially high-load process to be carried out in parallel to other game functions.
- Game server(s); these run the core game engine and provide scalability by running in parallel.
- Proxy server(s); these are responsible for all server-to-client communications and perform three main functions. Firstly, removing the need for game servers to de/encrypt and de/compress data; secondly, improving security and filtering out any harmful traffic (for example, Denial of Service attacks) by removing the direct connection between clients and game servers; thirdly, improving scalability by allowing new proxy servers to be created dynamically.
- A synchronisation server; a relational database can be a potential bottleneck, especially as the number of players increase, as frequent real-time access can considerably reduce the system performance. A synchronisation server can alleviate this issue by storing the game data in memory, allowing fast access. The database still needs to be updated but the synchronisation server can do this in the most efficient way possible (for example, in batches). Having an additional copy of the game data also helps with recovery from system failures.
- A 'world server'; with so many separate components, the world server is responsible for co-ordination – for example, the matching of clients to game servers, spawning new proxy or game servers, the provision of centralised functions such as chat.

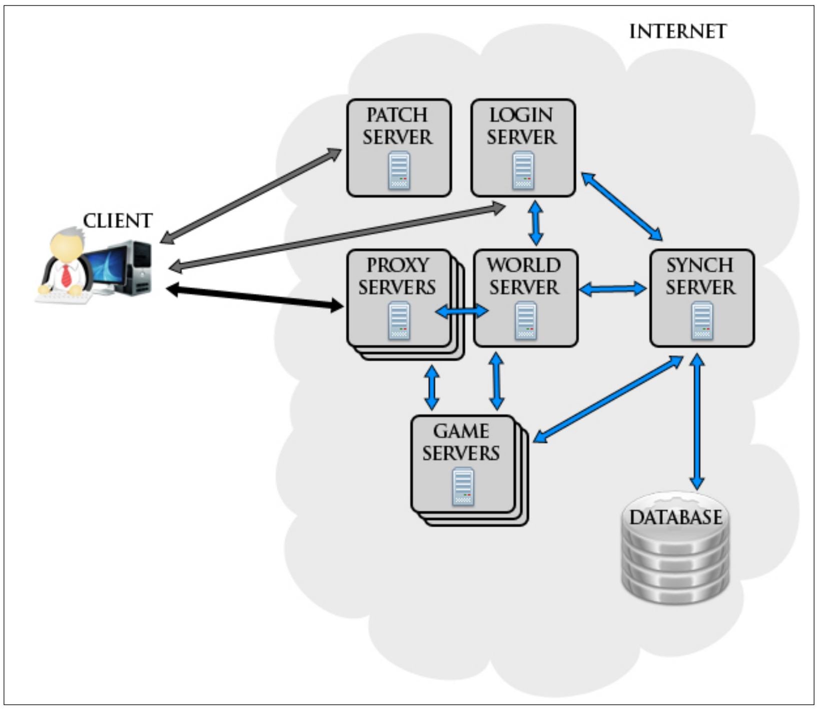


Figure 4: A distributed-function MMO architecture (Dieckmann, 2013)

Radoff 2007) proposes a very similar architecture but includes two additional elements:

- A single 'worldwide' chat server, enabling chat amongst all game clients, no matter which game server they are currently connected to.
- A web server, through which certain game-related data may be exposed (for example, rankings, group memberships, etc.).

Caltagirone et al. (2002) propose a C-S MMORPG architecture with several optimisations. These include, firstly, a single database stores the persistent world state but, for ease of access and to reduce any network 'bottlenecks', subsets of the database are stored in direct access memory on different servers, each of which can correspond to particular game world servers. This arrangement allows the game world to be divided amongst several game servers, each of which contains a geographical zone of the game world, thus providing scalability. Additionally, this arrangement also provides a certain degree of redundancy.

Secondly, the architecture provides for a separation of functions, both on the server and client sides. Server-side a module (the "Governor") administers server co-ordination, user authentication, player group functions, AI components, and network functions (e.g. en/decryption). This allows the core game functions to operate without the added computational load associated with the periphery functions. It also facilitates the alteration of individual components independently of each other.

A similar central control module (the "Mayor") exists client-side. This handles a number of functions, including the maintenance of the local game state, updated as necessary by server updates; the real-time processing of graphics; and the translation of player input into game state data and graphical representation. This arrangement enables the client to assume some of the processing burden which, again, improves network traffic and overall performance.

2.2.3 System architecture: Load balancing strategies

In order provide for scalability, an MMORPG architecture must identify the most efficient method for balancing the processing load between a number computing resources.

One common method amongst C-S architectures is the use of 'shards', a term originally associated with the MMORPG *Ultima Online* (derived, in fact, from the actual game lore). Shards are mirrors of the game world, each of which runs independently in parallel on one or more servers. Whilst allowing for an even distribution of players and computing resources, Drain (2011) points out that there can be significant disadvantages, not least of which is the dividing of the player-base into sub-communities, thereby limiting the opportunity for player-vs-player (PvP) play, and introducing the potential for skewing the shards' economic and political models.

An alternative to the traditional shard model is the single-shard model used by *EVE*, the space-based MMORPG, and described by Drain (2008) and Emilsson (2014). *EVE* maintains a single world for all of its players (approximately 500,000), consisting of over 5,000 star systems, each of which is run as a separate game process. *EVE* runs on hundreds of 'SQL' servers, all of which are connected to a central database server which maintains the game state. The mapping of star systems to physical servers is dependent on the star system population, so that a single server might house one particularly popular star system, or several low-population star systems. In order to keep track of which SQL server a player is using, and to facilitate player transfer from one server to another (i.e. travel between star systems), EVE uses a series of proxy servers to which the clients are connected.

Emilsson highlights a number of advantages of the single-shard: from a technical standpoint, not having to cope with the synchronisation and duplication issues inherent in moving data between databases; from a game experience standpoint, principally a more realistic evolution of both the economic and political systems and the development of a mature and complex, albeit less forgiving, 'social tapestry'.

However, there are also disadvantages: firstly, the high cost of purchasing, maintaining and upgrading servers in order to keep up with an expanding player-base; secondly, the potential bottleneck caused by using a single underlying database. Emilsson states that "running an MMO in a single shard introduces strains on system architecture, low-level runtime, databases, and operations."

Whether or not single or multiple shards are employed, a common model for balancing the processing load is to divide the game world into several geographical zones and to distribute these zones amongst the available servers. Clients will then connect to whichever server is hosting that part of the game world in which they are currently residing.

One example is the architecture described by Emilsson above. Another is proposed by Assiotis and Tzanov (2005), who highlight one of the main challenges associated with this method, namely how to synchronise, without a noticeable detrimental effect on performance, the transition of a player from one zone to another. Many games solve this issue by partitioning one zone from another, only allowing movement between zones through the use of artificial mechanisms such as a sea voyage or a teleport; however, this method introduces strict limitations on the design of the game world.

Assiotis and Tzanov propose a model that not only allows free movement between zones but also provides for events that occur close to zone borders and therefore affect multiple zones. They achieve this through the use of the Publish-Subscribe pattern, in which neighbouring servers are automatically subscribed to receive notification of events that occur within a specified border area.

For example, if a player *P* passes over the border then *AA*, the server that currently 'owns' *P*, will send an event message to *BB*, the new server, indicating that *BB* now has ownership of *P*; this message will contain all state information for *P*, which includes a list of all other players that are within interaction distance of *P. BB* will then contact *P*'s client, telling it to connect to *BB* from now on.

Variations of the geographical model exist in which load-balancing algorithms are used to attempt to dynamically balance the load between physical servers. Kim and Park (2013) cite research by Andrade and Corruble (2005) and Jinzhonh and Zhigang (2010) in which the game world is divided into fixed-size cells, multiples of which are hosted on a single server. When the load on a server reaches a certain level, cells can be transferred to a server with a lesser load. Jinzhonh and Zhigang propose an algorithm that selects the cell with the least interaction with its neighbouring cells for transfer to another server.

Alecu (2012) states that two key issues with dynamic load-balancing systems, such as those mentioned above, are: i) each server needs to be in constant communication with its neighbours in order to know which are available in case a transfer of cells is required; ii) the computational load dedicated to this process, can have a detrimental effect on core game functions. He proposes an architecture in which this burden, and other central functions, is assumed by a 'master' server which monitors the load of all game servers and spawns new servers as required, or redistributes the overall load in order to shut down unneeded servers during quite periods.

By definition, in P2P or hybrid architecture models, the load is distributed amongst the participating peers, with each client assuming responsibility for the immediate locality – area of interest (AOI) – in which the player is operating. This has the advantage of greatly reducing the need to maintain large numbers of servers, and also of being able to dynamically respond to the ebb and flow of concurrent player numbers. However, it also introduces problems: Firstly, the load is distributed but not balanced. Players in densely crowded areas of the game world have a much higher load than those in sparsely populated areas. Secondly, there is the potential for a significant degree of data duplication on different clients, and of the need to synchronise game state between clients. Again, this is exacerbated by high player density.

One area, reviewed by Fan et al. (2010), in which the load can be more evenly distributed is that of NPC hosting. NPCs are AI-controlled characters that are present in a game in order to provide additional interaction for players, or to contribute to the storyline, or to simply make the game world more realistic and immersive. However, they also represent a significant computational and network traffic load, which increases in relation to NPC numbers and AI sophistication.

Fan et al. outlines two main approaches that are used to handle the NPC load. Firstly, as proposed by Knutsson et al. (2004) and Iimura et al. (2004), the region-based approach, in which a 'super-peer' is given the responsibility for hosting all NPCs in a given geographical region. This removes the load from most clients but can result in a huge burden on the super-peer, especially as the super-peer is often not chosen for its suitability (i.e. processing capacity) but simply on a 'first come, first serve' basis.

Secondly, as proposed by Bharambe (2006), Yonekura et al. (2004) and Hu et al. (2008), the virtual-distance-based approach, in which an NPC is hosted by the player nearest to them in the game world. This spreads the NPC load more evenly and potentially reduces network traffic because the NPC will be hosted by the player it is most likely to interact with. However, once again, areas of high NPC density can result in a high computational load on the nearest player and, in games with high mobility, it may require much switching of NPCs from one host to another and much computation to decide upon the nearest (and therefore hosting) player.

Fan et al. (2010) proposes a third approach, heterogeneous task-sharing. In this approach, super-peers are used in a match-making role in which, when the game requires the creation of a new NPC, they select the most appropriate peer to use as a host, based on processing capacity. This not only ensures that the NPC load is distributed in a more efficient manner but also, as the host will remain with that peer until it leaves the game, reduces the need to transfer between hosts.

Hybrid systems can use elements of both C-S and P2P architectures to manage load balancing. Driel et al. (2011) describe a hybrid architecture, proposed by Jardine and Zappala (2008), in which a central server is responsible for maintaining the game state, and for electing a number of peers to be regional servers for a specific geographical zone. Peers are connected to a regional server, which is responsible for disseminating any movement-related updates amongst its connected peers; any non-movement related changes of state (for example, changes to player inventory) are handled by the central server, which then sends updates to the regional servers for dissemination.

Another hybrid model, mirrored game architecture, described by Cronin et al. (2004), uses multiple game servers, each of which contains an identical game world. Clients connect to any of the game servers, although the system can be configured to automatically assign the closest server, thereby reducing latency. The game servers are then connected to each other in a peer-to-peer relationship, using a high-speed VPN, and any state changes made on one server are communicated to all others. This architecture allows the balancing of overall load amongst server-peers, whilst still retaining many of the advantages of the C-S architecture, listed in *Section 5.2.1*. However, as discussed by Assiotis and Tzanov (2005), there are some disadvantages: Firstly, as player numbers increase, it becomes ever more difficult to maintain a consistent game state between all servers; secondly, servers may require considerable processing capacity in order to maintain a copy of the entire game world.

2.2.4 System architecture: Minimisation of network traffic

In order to reduce latency and improve game performance, it is essential to reduce the amount of processing overhead and bandwidth associated with network traffic.

One of the key concepts associated with this issue is that of interest management (IM) – how to present the player with the information that is relevant to him (i.e. his AOI) whilst minimising any non-relevant information. Fan et al. (2010) summarise three methods used to address this issue in P2P architectures.

Firstly, the spatial model, also known as 'aura-nimbus', in which the nimbus is the area within which an object can perceive other objects, and the aura is the area within which an object can interact with other objects: the breaching of a player's nimbus allows preparation to take place before the actual interaction occurs (within the player's aura). This model allows the fine-tuning of the state and event data sent to a player but can also result in additional network traffic between peers in order to maintain a spatial awareness of other objects. Fan et al. state that a variant of the spatial model, proposed by Buyukkaya, E and Abdallah, M. (2008) and HU, S-Y. et al. (2008), reduces this extra traffic by requiring peers to create Voronoi diagrams in order to perceive their relationship with their neighbours, each peer acting as a 'lookout' to warn its neighbours of the approach of any new objects. However, although this reduces network traffic, it adds an additional computational burden associated with maintaining the Voronoi diagrams.

Secondly, the regional model, in which the game world is divided into zones; using the Publish-Subscribe pattern, peers will be subscribed to receive updates on all other objects in any regions that currently intersect their AOI. This model reduces the computational overhead associated with the spatial model and allows a simpler communication structure (for example, the use of multicast) but, if the region size is less than optimal, can potentially result in the player receiving unnecessary updates.

Finally, the hybrid communication model combines elements from both of the models above. The game world is divided into regions and a super-peer is selected for each region which receives movement updates from each peer that enters its region. This allows the super-peer to maintain a spatial overview of its region, enabling it to ensure that communications are established between players that are likely to enter each other's AOI. This model is less complex than the pure spatial model, whilst nevertheless allowing the fine-tuning of traffic between peers; it also reduces the unnecessary additional traffic associated with the regional model. Its chief disadvantage, however, is that it places a high computational and communication load on the super-peer, so it may be necessary to implement some kind of dynamic region resizing in order to cope with areas of high player density.

Assiotis and Tzanov (2005) propose a C-S architecture that uses the Publish-Subscribe pattern in order to reduce the amount of network traffic exchanged between client and server; again the concept of AOI is employed to ensure that a client only receives relevant state and event data. In this model, the game world is divided into regions, each of which is controlled by a physical server, and each player has a defined AOI (which may vary from player-to-player). It is the responsibility of the server to ensure that each player is automatically subscribed to receive updates regarding objects and events that occur within its AOI. The Publish-Subscribe pattern is also used between servers to ensure that events that occur on a neighbouring server can be perceived by players close to the border.

Bharambe et al. (2008) propose a P2P architecture, *Donnybrook*, that attempts to reduce the amount of network traffic by using a novel adaptation of the AOI concept. *Donnybrook* takes advantage of the limited human capacity to simultaneously focus on multiple objects, in order to reduce the number of objects within a player's AOI about which they will receive updates. For each player, the system compiles an 'interest set' of five players that they are most interested in, based on proximity, aim (i.e. orientation), and how recently they have interacted. A player will only send a real-time update to another player if they are included in that player's interest set. Players within the AOI that are not in the interest set are represented by bots controlled by the game's AI but guided by less frequent updates from the player they represent. In this way, the amount of network traffic is reduced, particularly for high-density occurrences, such as large battles.

The architecture proposed by Kim and Park (2013) attempts to reduce the amount of data transferred between servers by anticipating likely interaction between players. They suggest that, given the importance of player interaction in MMORPGs, it would be sensible to place adjacent players (i.e. those who fall, or are likely to fall, within each other's AOI) on the same server where possible, thus reducing data transfer between players on different servers. They propose a system for selecting an appropriate server when the player logs in, based on the adjacency of other active players.

Using the game *TailPlanet* (developed by *Connected Dreams*), Alecu (2012) proposes a similar system that attempts to address the issue of data transfer between servers, resulting from players on different servers interacting with each other. He introduces a function that monitors the ‘closeness’ of players based on previous interactions, taking into account factors such as the amount of chat interaction between players, the amount of game-related interaction (i.e. time spent in each other’s AOI), and the length of interactions as compared with the overall length of game sessions. When a player logs in, the system will use this data to place them on the server most likely to facilitate interactions, based on which other players are currently active. The function is also used in-game when a user on one server wishes to interact with a player on a different server. In this case, the function is used to determine which player should be moved, and whether any other players should also be moved in anticipation of future interaction. When tested with *TailPlanet*, use of the closeness function showed a 12% decrease in the amount of data transferred between servers.

Other approaches at reducing network traffic involve changes to the traditional practices associated with communication over networks. McKnight et al. (2012) focus on games that divide the game world into regions distributed across multiple servers; many of these games, in order to facilitate cross-border perception and movement, use overlapping border zones in which all objects are duplicated on each server. In order to maintain game state consistency, if the state of an object in a border area is altered, it is necessary to obtain an inter-server lock on the object, a process that not only requires the implementation of a complex system component but also involves additional inter-server network traffic.

McKnight et al. propose a C-S architecture (*Gendu*) that does not use inter-server locks, thereby simplifying the design by removing an entire component process, and increasing efficiency by reducing inter-server communication. There are a number of requirements: firstly, a system to determine ‘jurisdiction’ over mutable (i.e. state-changeable) game objects; secondly, an asynchronous remote-write protocol in which all changes to an object’s state must occur on the server with jurisdiction; thirdly, the use of only prescribed types of write operation. Writes must be atomic operations, and are only regarded as successful when the value is read back from the primary source (i.e. the server with jurisdiction). Also, in order to avoid one update being asynchronously overwritten by another, writes must use delta values (where modifications are made in relation to an existing value).

McKnight et al. tested *Gendu* using both single server and multi-server configurations and were able to demonstrate that their lockless architecture could successfully support a game world divided amongst multiple servers.

2.2.5 System architecture: Data storage

In order to provide for a persistent game world, a suitable method must be found for the storage of game state data. This function can be handled in very different ways, depending on the type of architecture being used.

Traditionally, C-S architectures utilise a relational database management system (RDBMS); this allows data to be modified in an ACID-compliant manner (atomicity, consistency, isolation, durability), whilst also providing a very powerful tool for obtaining data related to system usage and performance. The database being centrally controlled also provides additional security and reduces the opportunities for cheating.

However, the database can also be a performance weak point for an MMORPG; although relational databases have been designed for the efficient throughput of transactions, they often find it hard to cope with the sheer volume generated through a fast-moving MMORPG with thousands of concurrent players. Discussing the space-based MMORPG *EVE*, Emilsson (2014) states that “the main bottleneck that we have had to overcome is I/O performance of database storage.”

A common method used to mitigate the database bottleneck is that of employing a number of interim servers, which cache the data in direct-access memory, greatly speeding up the transactions times. In the architecture proposed by Dieckmann (2013), a ‘synchronisation’ server is used to store the game data in memory, removing the requirement for direct real-time access to the database by clients (see *Figure 4*). Other advantages of this method are, firstly, that having additional copies of the game data can help with recovery from system failures and, secondly, it enables scalability by spawning additional data servers.

This method still requires that the central database, being the authoritative source of game state data, be regularly updated but this can be done in a more efficient manner (for example, at optimum intervals in batches).

There are variations of this method. Caltagirone et al. (2002) propose that, as each game server hosts a specific region of the game world, the data server connected to that game server need only contain data relevant to that region. Similarly, each ‘SQL’ server used by *EVE* only contains data for those star systems that it hosts (Emilsson, 2014).

The architecture proposed by Alecu (2012) divides the data in a different manner, comprising of two complimentary database layers. One layer allows direct access to the database and contains information that is rarely changed (for example, player account information and state data for immutable game objects); the other layer keeps the data in memory and contains game state data that is frequently changing. In this way, the fast-access memory is more efficiently targeted and utilised.

Another factor to consider is the type of database to use. Traditionally, the type most commonly employed in computer systems is the relational database, which allows data to be stored in separate tables, according to certain criteria, and between which relationships are defined, based on key values in each table. Structured Query Language (SQL) can be used to interact with the database for the purposes of insertion, deletion, or data selection.

However, there are a number of alternative types, broadly described by the umbrella term ‘NoSQL’, meaning non-relational. The various types of NoSQL database include document-oriented, key-value store, and object. There can be a number of advantages for NoSQL databases, including that they sometimes perform certain tasks faster than relational databases, or that their structure may more closely match the data structure used in a computer program. There are also inherent disadvantages, however, such as the lack of sophisticated querying languages, the lack of ACID support for transactions, and the fact that there will usually be a steep learning curve associated with their adoption.

In his proposal for a C-S architecture, Muhammad (2011) evaluates three different types of database, namely *MySQL* (an open-source relational database), *CouchDB* (a document-oriented database), and *Riak* (a key-value store database). Each database was tested on a single database server by using scripts to perform select, insert, and update operations, with the number of transactions per second being recorded against the number of players connected. Results showed that, with 200 concurrent players, both of the NoSQL databases out-performed *MySQL* - *CouchDB* by approximately 7.6% and *Riak* by approximately 23%.

Diao et al. (2014), identifying the challenges associated with data storage in MMORPGs, studied the use of a cloud-based, distributed NoSQL database, Cassandra, to provide data storage for an MMORPG, focussing on the areas of scalability and performance. Cassandra, a wide-column store DBMS, has the advantage of not requiring the use of the join operations needed in a relational database; however, some data duplication is required in order to ensure that not only could all data associated with an object could be retrieved in a single atomic operation, but that certain component parts could also be retrieved separately. Tests showed that increasing the number of nodes led to a marked increase in the number of clients that could be supported, from 600 clients in a 1-node configuration to 1500 in a 3-node configuration; operation times (i.e. for read/write) were also improved.

Alqwbani et al. (2014), performed a similar study of Cassandra and reported analogous results. They attributed this to certain inherent advantages of NoSQL databases: i) they lack the high overhead required for distributed relational databases in order to maintain consistency in the schema; ii) they have no requirement for ACID constraints; iii) they have no requirement to link separate data with queries.

They also identified two further advantages: firstly, when using a relational database, any changes to the table structure will often require extensive refactoring of game code. This is not so with a NoSQL database, which can handle highly varied (or unique) objects without any need to define a schema. Secondly, NoSQL databases tend to use simpler interface protocols, as there is no need for SQL binding.

Zeigler (2006) suggests that object databases may be more suited to the demands of an MMORPG and may result in an improvement in performance. The reason for this is that the structure in which data is stored in a relational database does not map efficiently onto the data structure used in a game's program code. Zeigler states that "SQL requires all data to be placed in simple tables … However, game data is very structured and hierarchical." In order to access the data for an object, the programmer has to create SQL statements that define relationships for data held in different tables; quite often this requires the creation of an SQL layer between the game server and database, which results in additional latency. With an object database, however, the data structure reflects the code structure which not only makes for faster retrieval but also automatically handles object-oriented concepts such as inheritance.

The configuration of the database can also have an effect on performance. Distributed databases are designed to operate on multiple hosts, which enable them to improve performance by not only taking advantage of the increased processing capacity, but also by allowing them to be located closer to separate (and possibly geographically distant) communities of end-users, thus reducing latency. Muhammad (2011) evaluated the distributed database, *Riak*, comparing the number of transactions per second for single node and two-node configurations. Results showed that the single node was able to only satisfy 1,200 transactions per second, whereas the two-node configuration was able to satisfy 1,500 per second.

Because of the lack of a permanent server-side structure, P2P architectures have to handle the issue of data storage in different ways from C-S. One architecture, proposed by Buyukkaya et al. (2009), makes a distinction between the storage of game data and the responsibility for determining who receives updates on changes in state. The storage of data is achieved through the introduction of a structured database layer, based on a distributed hash table (DHT), which evenly distributes the data amongst all connected peers, regardless of where they currently reside in the game world. Peers then use Voronoi diagrams to gain a spatial overview of all objects and players in their game world neighbourhood. If an object falls inside the Voronoi cell of a particular peer, that peer determines whether the object is within the AOI of any of its neighbouring peers, then sends this list of 'interested' peers to the objects host peer (as specified in the DHT). The hosting peer then knows who to send any state updates to.

Fan et al. (2010) point out that a potential weakness of a system such as that proposed by Buyukkaya et al. (2009), above, is that the read/write speed for data stored in a P2P structured database layer is slow and results in increased latency. Fan et al. (2010) mention another P2P architecture, proposed by Iimura et al. (2004), that attempts to solve this problem by using a system of super-peers to store volatile data in memory, allowing updates to occur much more efficiently. The non-cached, distributed data storage layer can then be utilised for backup purposes.

3. METHODOLOGY

As described in *Section 1.1 Aims and objectives*, the project was structured into two main phases: design and implementation.

The design phase addressed, firstly, the development of a core game model, specifying the basic game classes, interactions, rules, mechanisms, and victory metrics; secondly, the design of a game engine enabling players to perform the functionality outlined in the model, and taking into account the subsequent introduction of client-server architecture.

Key considerations to be addressed during the implementation phase were, firstly, to ensure modularity through the creation of clear interfaces and the use of standard and well-documented protocols; secondly, to ensure scalability through the use of components and architecture that would ensure high performance during periods of high usage.

The implementation phase also included a planned period of testing and code review and the creation of a basic client in order to facilitate the implementation and testing of the core game engine.

3.1 System development

It was decided that system development would proceed in the following stages:

3.1.1 Game model: The design of a game model that fulfils the functional requirements specified in *Section 1.3 Requirements and specification* (see *Figure 5: Use case diagram*). This model should attempt to minimise complexity by reducing the size of the game world, and the number of game class attributes and mechanisms; however, the model should be expandable, allowing for additional concepts to be added at later stages.

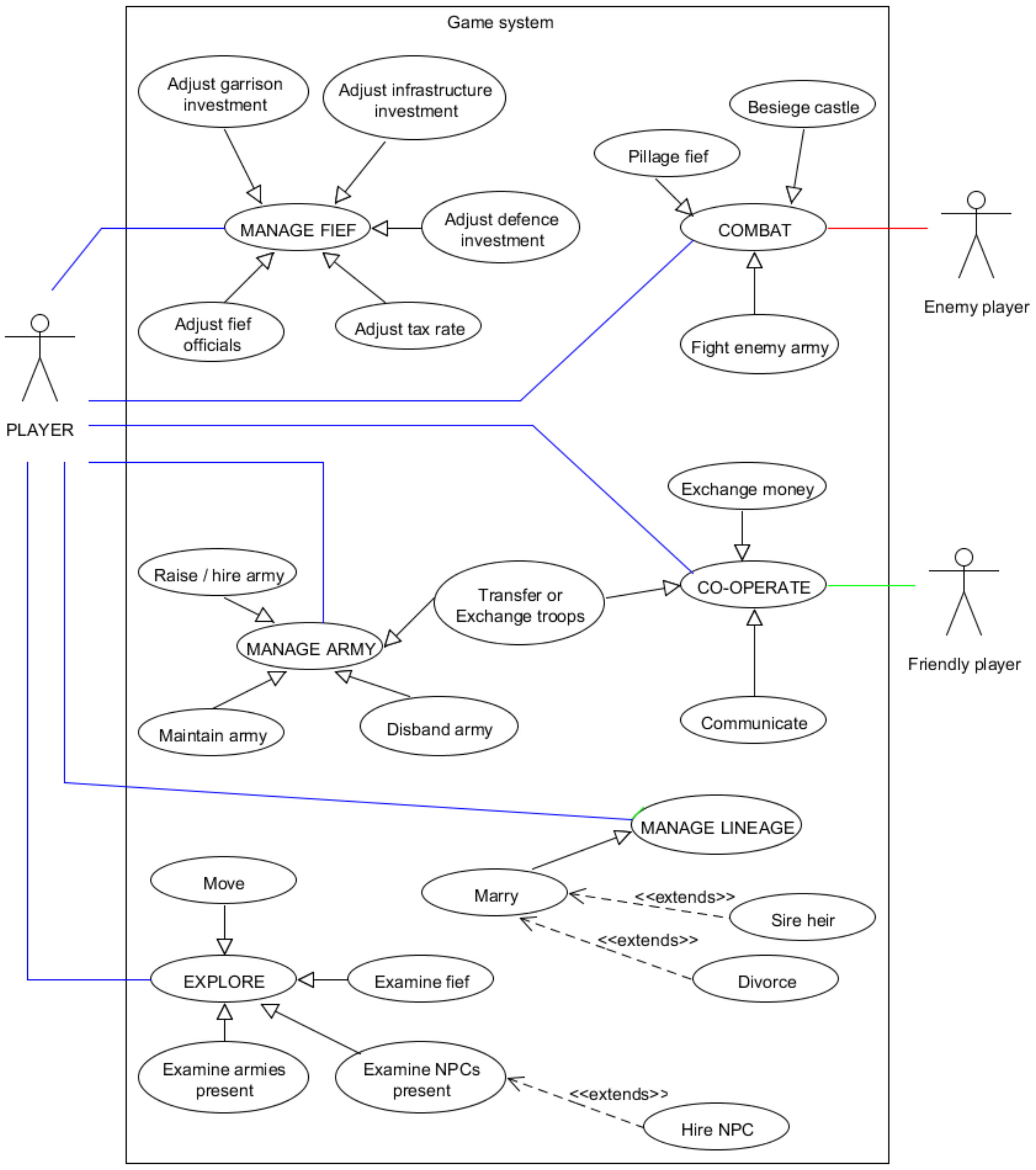


Figure 5: Use case diagram showing main player actions

3.1.2 Game engine: The design and production of a simplified game engine that implements the game model above.

A basic user interface (text-based or button-based), should be developed, separate from the game engine and accessed through a test client; at this stage, the purpose of the user interface is simply to allow testing of the game engine. However, as mentioned in *Section 1.3.2: Extra-functional requirements*, provision should be made for the future addition of a full graphical user interface (GUI). One possible development engine that might be used for the implementation of a GUI is *Unity 3D*, which uses C# to create scripts.

Game content will be expanded in future stages; therefore, all interfaces and protocols should be clearly defined and documented. Where possible, the game engine should make use of standard definitions for the communication of game data; for example, the GEDCOM specification could be used to communicate and display PC genealogical data.

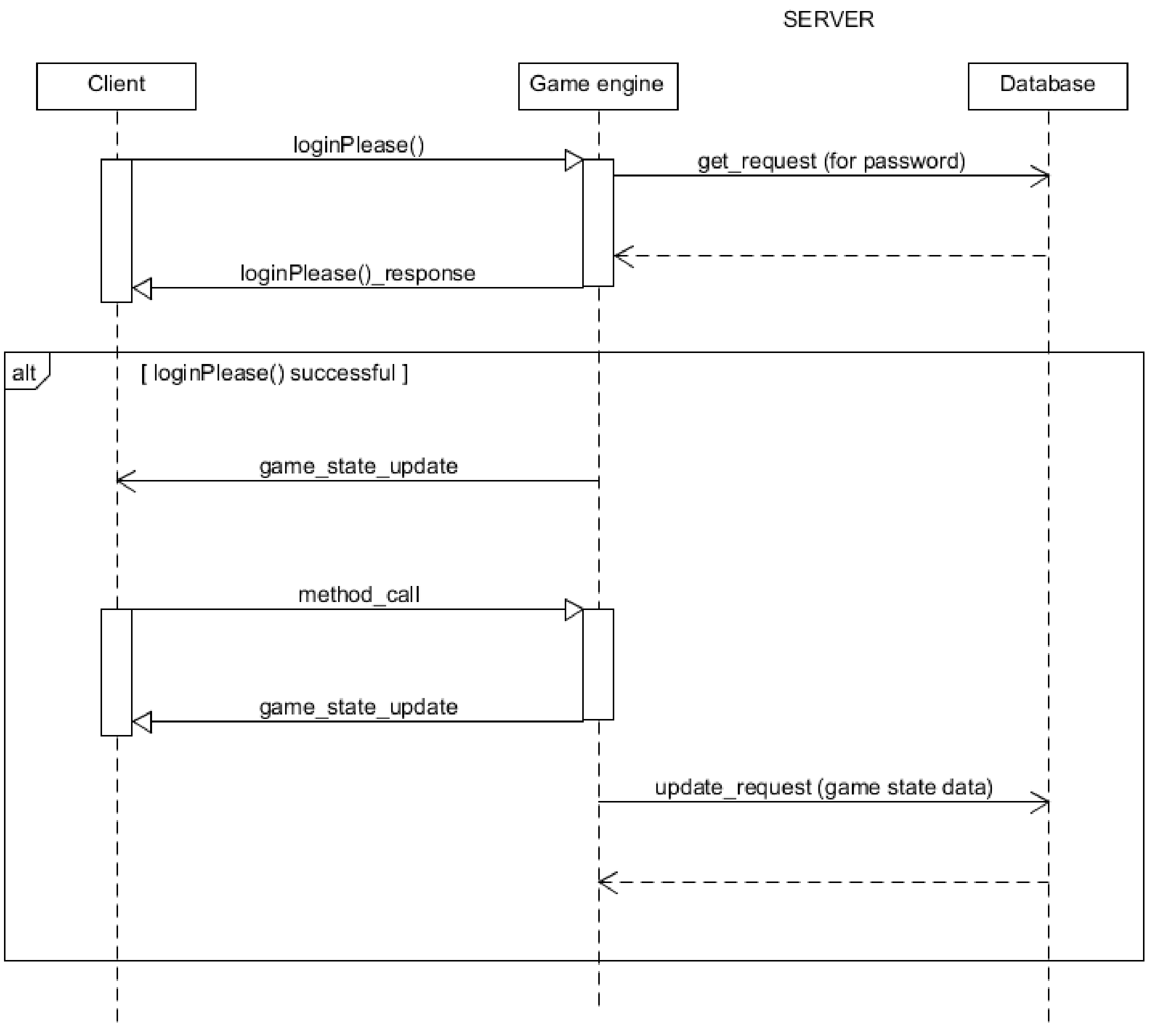


Figure 6: Sequence diagram showing an example of client-server communication

3.1.3 Game system architecture: Most importantly for this project, the design and implementation of an underlying game system architecture that best achieves low latency during periods of high usage. A number of decisions need to be made in order to find the best methods and structure to achieve this goal; the decisions include:

- In what format and structure to store the game data. As discussed in *Section 2.2.5* of the *Literature review*, there are a number of different options including a relational DBMS, any one of the variety of NoSQL databases, or other formats (such as XML files).

  In the context of both the functional and extra-functional requirements (see *Sections 1.3.1.3 and 1.3.2.2*), the initial choice of DBMS was *Riak*. This is due to its speed and basic query support, and because it is also a distributed database, which may help spread the data load and reduce latency – see Muhammad (2011). Should Riak prove unsuitable, another possibility is a relational database such as SQLite, a fully featured relational DBMS with a lower performance overhead than many of its competitors.

- How best to achieve load balancing. As discussed in *Section 2.2.3* of the *Literature review*, there are a number of options, including the division of the game world into zones (hosted on separate servers), the replication of the whole game world, or the use of P2P architecture to spread the load between the players computing resources.

  Regarding, the basic architectural model, it was decided that the project would use the client-server approach, as it provides for simpler design and testing, and is beneficial to security and control.

- How best to reduce the load on the game server through abstraction of functions. The game engine will include many distinct functions, some of which can be separated, facilitating modularity, and potentially distributed, improving scalability and core game engine performance by running specific resource-heavy functionality on a dedicated server (for example, see *Figure 4* in *section 2.2.2* of the *Literature review*).

  In order to simplify the initial system architecture, it was decided to investigate the possibility of 'outsourcing' these functions; for example, the use of Facebook Login for account authentication, and the use of a third-party chat or bulletin board system for inter-player communication.

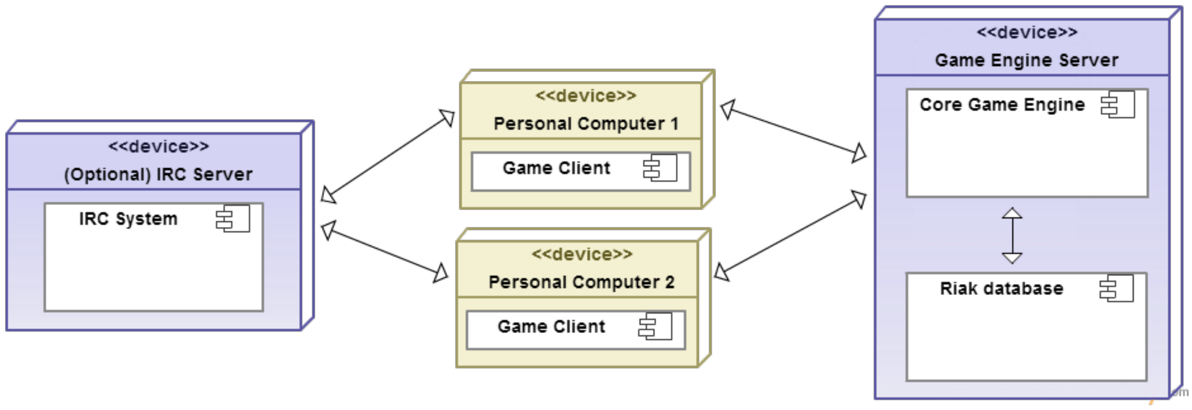


Figure 7: Deployment diagram showing initial system architecture

3.2 System evaluation

The game engine should be evaluated in respect of two main areas:

- Firstly, to ensure that it fulfils the functional requirements outlined in *Section 1.3.1*. This testing should utilise the basic user interface (see *Section 1.1.1*) and could, for example, take the form of a scripted sequence of actions carried out by a number of human participants.

- Secondly, to ensure that it fulfils the extra-functional requirements outlined in *Section 1.3.2*. The performance of the game engine should be evaluated by measuring the degree of latency at varying usage levels. This requires a low-latency network environment, in order to isolate latency caused by game system architecture. Testing requires two main components: i) automated test agents (simple 'bots') to provide the driver for system load, simulating usage levels; ii) timers, built into game actions and accessed through the test client, to perform the actual latency measurement.

Testing should be undertaken both concurrently with coding and in a dedicated period towards the end of the implementation period.

3.3 Environment and software

The environment chosen for hosting the game system was:

- *Linux* 2.6.32-358.18.1.el6.x86_64 running *CentOS* release 6.4 (Final). This particular Linux OS was chosen primarily because of its reputation as a stable platform but also because it supports the chosen DBMS (Riak), and is currently used on machines in the Linux labs in the School of Mathematical & Computer Sciences, Heriot-Watt University. This should ensure, firstly, flexibility as regards to hosting and, secondly, familiarity with the platform for any participants in future projects to expand the system.

The following local environments and software tools were chosen for development:

- *Linux* 2.6.32-358.18.1.el6.x86_64 running *CentOS* release 6.4 (Final) and *Windows 7 Home Premium* 6.1.7601 (Service Pack 1 build 7601). The dual OS approach is necessary to ensure that the system meets the specified extra-function system requirements (see *Section 1.3.2.1*). It also allowed the game system to be initially hosted in a Linux environment with access to the *Riak* DBMS, prior to the development of the client-server configuration.

- *C#* 4.5. When deciding on which programming language to choose, particularly with the dual OS requirement in mind, two languages stood out – *Java* and C#. Both are fully-featured, object-oriented languages that boast a well-developed catalogue of supporting libraries. *Java* is platform-independent but C# can also run on other platforms through the use of the *Mono* system (see below). Crucially, with a view to the possible future development of a GUI for the game system, C# is used by *Unity 3D*, a cross-platform game creation system built using *Mono*. For this reason, C# was chosen for this project.

- *CorrugatedIron* 1.4.2 C# library. At the time of development, this was the only library for connecting C# to *Riak*, and provides easy access to the basic *Riak* functions (search, retrieve, write) required by project. It makes use of *JSON.NET* 4.5.10 library for serialising object data, thus minimising the need for the creation of custom methods; this also introduces the possibility of using JSON as the format for data exchanged between the game engine client and server.

- *QuickGraph* 3.6.61119.7 C# library. An optional requirement of the user interface is the provision of a hexagon map, which would entail the creation of a graph structure within the program. *QuickGraph* is one of several libraries that facilitate the creation of graphs, and is widely regarded in C# community as being the simplest to use, whilst providing a number of useful graph functions (for example, a shortest path function based on Dijkstra's algorithm).

- *Microsoft .NET* 4.5. Although primarily designed for the Windows OS (to provide cross-language inter-operability), this software framework can be supported on other platforms through the use of *Mono*. It is required by the C# programming language.

- *Mono* 3.4.0 As previously mentioned, it was a requirement that the game system should be executable in both Microsoft Windows (version 7 onwards) and any Linux-based OS. *Mono* is an open source implementation of Microsoft's .NET Framework aimed at allowing developers to write cross-platform applications.

- *MonoDevelop* 4.2.2 and *Xamarin Studio* 4.2.5 (build 0). *MonoDevelop* is an open-source IDE, created as a part of the Mono project for use in developing and debugging applications designed to be run in Mono. In a project that requires developing across different platforms, *MonoDevelop* and its sister IDE, *Xamarin Studio* (the Windows implementation), provide additional continuity.

- *Riak* 1.4.8

- *Microsoft Visual Studio 2010* 10.0.40219.1 SP1. Developed specifically for use in the creation of .NET applications, MS *Visual Studio* is recognised as a very strong IDE for C# projects and would be the obvious choice for this project should *MonoDevelop* prove unsuitable. Additionally, it is also used in PC labs in the School of Mathematical & Computer Sciences, Heriot-Watt University, and would thus ensure familiarity for any participants in future projects to expand the system.

- *Git* 1.9.0.msysgit.0 (Windows), *Git* 1.7.1 (Linux) and *GitHub*. *Git* is a reliable, well-supported, cross-platform version control system (VCS). In combination with *GitHub* (the online repository for software projects using *Git*), it provides a variety of functions to the software developer including the ability to keep track of changes to their code, create separate development branches, and provide easy access to the current code to all interested parties. *Git* was chosen over rival VC systems such as *Subversion* because of its decentralised nature, which ensures that changes can be made to the local copy of the repository in situations where Internet connectivity is not guaranteed.

- *UMLet* 13.1. An open source utility for the creation of UML diagrams, containing templates for the most commonly used types, and allowing diagram export in a number of formats.

- *WinSCP* 5.1.4 (Build 3020) and *PuTTY* 0.63. These open source utilities provide FTP (*WinSCP*) and telnet/SSH (*PuTTY*) functionality should it be necessary to remotely access any hosts based at Heriot-Watt University.

4. PROJECT MANAGEMENT

4.1 Timeline

The project schedule was drawn up as follows (see *Figure 6: Project Gantt chart*):

Game model (Tuesday 8th – Friday 18th April)

- Classes and attributes
- Mechanisms (i.e. methods) and formulas
- Victory criteria
- User interface displays and functions

System model (Saturday 19th – Friday 25h April)

- Identify game engine components, including those that can be 'outsourced' (for example, login and communication)
- Identify required data metrics
- Select and design data storage
- Select load balancing method
- Select/design communication protocols
- Identify security requirements/mechanisms

Implementation – prototype (Saturday 26th April – Friday 23rd May)

- Identify which subset of game components are to be included in prototype
- Create game objects and mechanisms, using locally stored (or coded) data
- Create user interface (test client)
- Deliver game prototype

Implementation – architecture and advanced features (Saturday 24th May – Friday 27th June)

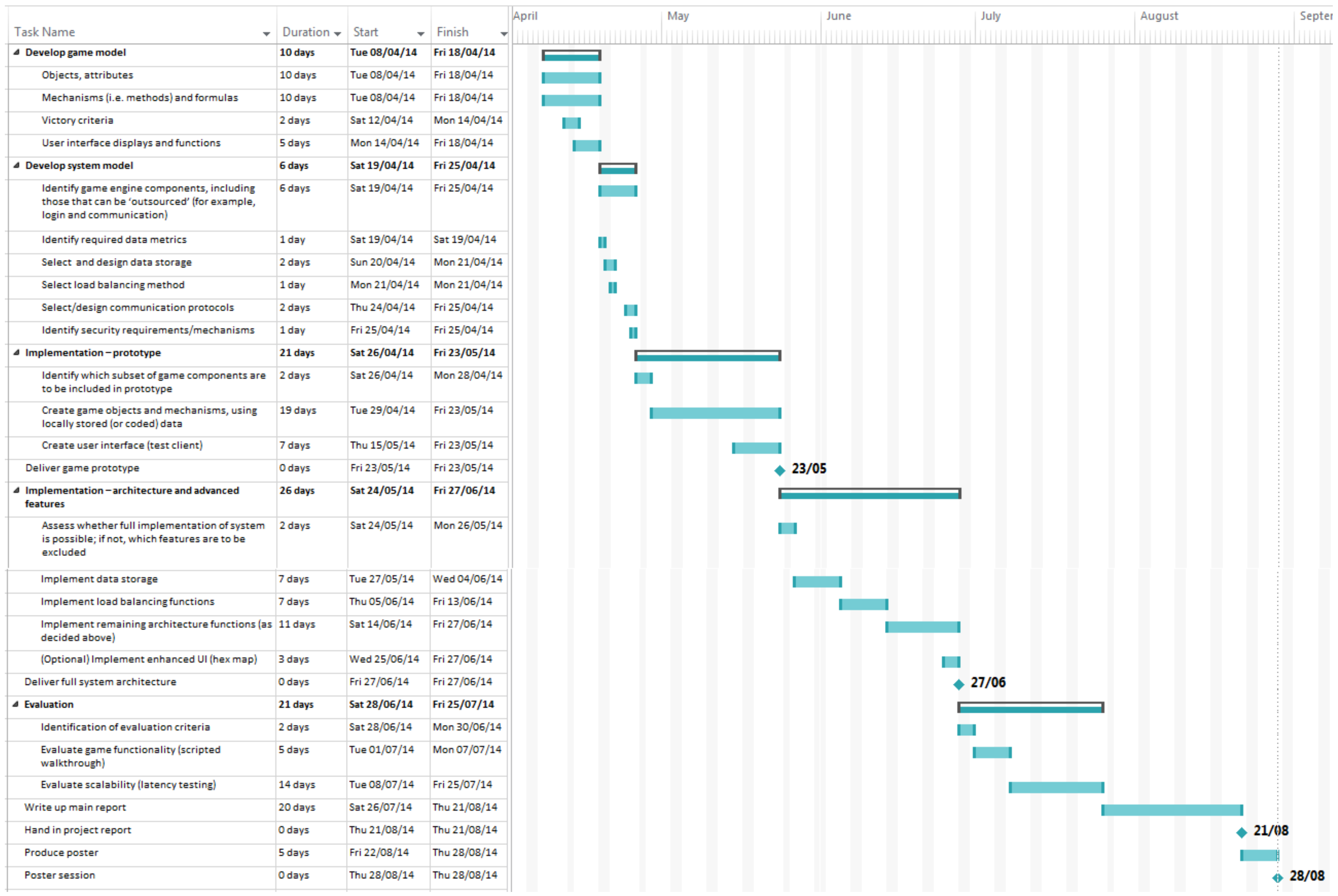
- Assess whether full implementation of system is possible; if not, which features are to be excluded
- Implement data storage
- Implement load balancing functions
- Implement remaining architecture functions (as decided above)
- (optional) Implement enhanced user interface (with hex map)

Evaluation (Saturday 28th June – Friday 25th July)

- Identification of evaluation criteria
- Evaluate game functionality (scripted walkthrough)
- Evaluate scalability (latency testing)

Write up and poster presentation (Saturday 28th June – Thursday 28th August)

- Main report
- Poster



Task Name	Duration	Start	Finish
Develop game model	**10 days**	**Tue 08/04/14**	**Fri 18/04/14**
Objects, attributes	10 days	Tue 08/04/14	Fri 18/04/14
Mechanisms (i.e. methods) and formulas	10 days	Tue 08/04/14	Fri 18/04/14
Victory criteria	2 days	Sat 12/04/14	Mon 14/04/14
User interface displays and functions	5 days	Mon 14/04/14	Fri 18/04/14
Develop system model	**6 days**	**Sat 19/04/14**	**Fri 25/04/14**
Identify game engine components, including those that can be 'outsourced' (for example, login and communication)	6 days	Sat 19/04/14	Fri 25/04/14
Identify required data metrics	1 day	Sat 19/04/14	Sat 19/04/14
Select and design data storage	2 days	Sun 20/04/14	Mon 21/04/14
Select load balancing method	1 day	Mon 21/04/14	Mon 21/04/14
Select/design communication protocols	2 days	Thu 24/04/14	Fri 25/04/14
Identify security requirements/mechanisms	1 day	Fri 25/04/14	Fri 25/04/14
Implementation – prototype	**21 days**	**Sat 26/04/14**	**Fri 23/05/14**
Identify which subset of game components are to be included in prototype	2 days	Sat 26/04/14	Mon 28/04/14
Create game objects and mechanisms, using locally stored (or coded) data	19 days	Tue 29/04/14	Fri 23/05/14
Create user interface (test client)	7 days	Thu 15/05/14	Fri 23/05/14
Deliver game prototype	0 days	Fri 23/05/14	Fri 23/05/14
Implementation – architecture and advanced features	**26 days**	**Sat 24/05/14**	**Fri 27/06/14**
Assess whether full implementation of system is possible; if not, which features are to be excluded	2 days	Sat 24/05/14	Mon 26/05/14
Implement data storage	7 days	Tue 27/05/14	Wed 04/06/14
Implement load balancing functions	7 days	Thu 05/06/14	Fri 13/06/14
Implement remaining architecture functions (as decided above)	11 days	Sat 14/06/14	Fri 27/06/14
(Optional) Implement enhanced UI (hex map)	3 days	Wed 25/06/14	Fri 27/06/14
Deliver full system architecture	0 days	Fri 27/06/14	Fri 27/06/14
Evaluation	**21 days**	**Sat 28/06/14**	**Fri 25/07/14**
Identification of evaluation criteria	2 days	Sat 28/06/14	Mon 30/06/14
Evaluate game functionality (scripted walkthrough)	5 days	Tue 01/07/14	Mon 07/07/14
Evaluate scalability (latency testing)	14 days	Tue 08/07/14	Fri 25/07/14
Write up main report	20 days	Sat 26/07/14	Thu 21/08/14
Hand in project report	0 days	Thu 21/08/14	Thu 21/08/14
Produce poster	5 days	Fri 22/08/14	Thu 28/08/14
Poster session	0 days	Thu 28/08/14	Thu 28/08/14

Figure 8: Project Gantt chart

4.2 Risk assessment

Main risks for the project are identified in *Table 2*, below.

Risk	**Impact**	**Likelihood**	**Resolution**
Project is too big or complex	Major	Possible (40-69%)	Simplify data (game world and object attributes); reduce functionality (game model & decisions)
Data storage is not stable or does not support all functions	Moderate	Unlikely (10-39%)	Switch to alternative data storage solution; ensure data can be ported into another system to perform missing functions
MonoDevelop IDE is not stable or does not support all functions	Moderate	Unlikely (10-39%)	Switch to alternative IDE system (e.g. MS Visual Studio); switch to alternative programming language (e.g. Java) in order to meet OS requirements
Project schedule negatively impacted by issues related to my employment	Moderate	Possible (40-69%)	Reduce project workload by simplifying data and/or reducing functionality; negotiate project extension
Loss of host or development platform	Major	Rare (1-10%)	Move to alternative platform using most recent data backup; negotiate project extension
Temporary loss of project author (e.g. due to illness)	Major	Rare (1-10%)	Reduce project workload by simplifying data and/or reducing functionality; negotiate project extension
Loss of project supervisor (e.g. due to change of employment)	Major	Rare (1-10%)	Arrange for an alternative project supervisor
Poor developer knowledge/capability leads to delays	Major	Unlikely (10-39%)	Reduce project workload by simplifying data and/or reducing functionality; provide additional developer training; negotiate project extension

Table 2: Risk assessment

4.3 Professional, legal, ethical, and social issues

4.3.1 Professional issues

As information technology professionals, it is incumbent upon the authors of this research to ensure that the project is implemented in a professional manner, according to good practice; this involves employing approved methodologies and displaying technical competence. The authors should also accept full professional responsibility for their work and any consequences resulting thereof.

As the programme under which this project is being undertaken (MSc Advanced Internet Applications) is accredited by the Chartered Institute of IT (BCS), attention should be given to that body's *Code of Good Practice*, particularly those codes pertaining to research. This document includes recommendations on practices such as project management and data security.

As a representative of Heriot-Watt University (both as a student and an employee), it is important to ensure that no harm comes to its reputation as a result of unacceptable standards of professionalism. In this respect, special attention should be paid to the Heriot-Watt University *Code of Good Practice in Research*; for example, it is considered unprofessional not to acknowledge the role of any collaborators or other participants who assist or support the research. It is also unprofessional not to keep clear and accurate records of any results.

4.3.2 Legal issues

In accordance with the *Copyright, Designs and Patents Act 1988*, it is essential that the authors of this project ensure that the use of, or reference to, intellectual property belonging to other individuals is correctly acknowledged.

It is important to ensure that the use of any software programs comply with their licensing requirements. For this reason, a special effort will be made to, where possible, make use of software that is open source or subject to free distribution under other licensing schemes (for example, Creative Commons).

As the project involves the creation of computer software, it is also necessary to give consideration as to which license will be used to govern its use and distribution. In order to allow as much freedom as possible, it has been decided to use the OSI (Open Source Initiative) approved **MIT license** (details: http://opensource.org/licenses/MIT); this basically allows any kind of use providing the original author is attributed, whilst ensuring that the original author is not held liable for any subsequent alterations or usage.

In accordance with the *Data Protection Act 1998*, it is required that any personal data stored in the game system is kept safe and secure, is accurate, and is only used for the limited and specific purposes for which it was acquired. In a similar vein, the BCS *Code of Good Practice* also encourages its members to protect the confidentiality of any private data used in research.

As this project does not involve research techniques that would require its collection, the amount of personal data stored in the system is likely to be very limited (for example, email addresses used as login names); nevertheless, care will be taken to protect any data from unauthorised access, and to provide a facility for players to delete their account, removing any associated data. Regarding accuracy, people registering to play the game will be warned that they are responsible for the accuracy of any data they input.

The *Computer Misuse Act 1990* covers the "securing [of] computer material against unauthorised access or modification" (Great Britain, Computer Misuse Act 1990). It is important, therefore, to ensure that the project does not result in access to data that is unauthorised by its owners, and that it does not employ software whose functionality can result in the committing of a crime.

4.3.3 Ethical issues

Information Technology professionals are often accused of being fixated by functionality, often at the expense of any ethical issues. In this context, therefore, it is even more important to give serious consideration to potential ethical issues before the practical stage of the project gets underway.

The Heriot-Watt University *Code of Good Practice in Research* makes specific reference to ethical considerations in research projects that involve human participants or animals. Of more relevance to this project, however, is the broader encouragement that all research be conducted in an honest fashion. Care will be taken, for example, to ensure that results are accurately reported, supported by primary data, and not in any way engineered in order to obtain a particular outcome.

The BCS *Code of Good Practice* also encourages ethical behaviour in research, asking its members not to take part in any research that is detrimental to society or the public, and to be aware of (and take responsibility for) the potential for misuse of research outcomes.

4.3.4 Social issues

One of the main aspects of an MMORPG is the element of interaction between players; this includes interaction controlled by game mechanics and enabled through the game interface, and free flowing communication, often enabled through chat systems. Where such interaction exists, therefore, there will always be the possibility of its misuse.

The game system created for this research project will attempt to reduce the opportunity for any form of in-game bullying of players by introducing the player-assigned roles of, firstly, system administrator and, secondly, faction heralds. Between them these roles will be responsible for administering the game system, and advising (and mediating between) players. Where necessary, they will have the authority to remove a player from a game, or even to ban a player from the system entirely.

It is also important to ensure that the game contains no content that can be construed as offensive. For example, a French person may take offense at the notion that English troops in the game are considered to be of higher quality than their French counterparts. Care must be taken, therefore, to maintain historical accuracy in order to counter any such claims of bias.

4.4 Project management methodology

As much as was possible in a development team consisting of a single person, it was decided to adopt an agile approach, albeit with some necessary modifications to allow for the particular circumstances of the project. Key features of the agile model used for the project include:

- An iterative approach, enabling an incremental focus on specific sub-sets of functionality, and avoiding the possibility of becoming over-burdened by long-term planning.

- Frequent communication: it was decided that regular face-to-face meetings be held between student and tutor for the duration of the project, enabling rapid feedback during each iteration. Due to the particular circumstances (both student and tutor are in full-time employment) the interval between meetings was set at two weeks, longer than is normal in the agile model. Communication by other means, chiefly email, was also encouraged during inter-meeting intervals. Additionally, the use of *Git* and *GitHub* allowed easy access for all project members to the current code.

- An adaptive approach: although key milestones were specified in the Project plan (see *Section 4.1*) it was recognised that these were likely to change due to emerging circumstances and that, therefore, a flexible attitude was advisable and should be supported through frequent communication between student and tutor.

- Testing: the agile model promotes the practice of testing in parallel with coding, rather than a dedicated testing period towards the end of the project, as is typically seen in the waterfall model. In this particular case, it was decided to combine both approaches by carrying out testing concurrently with coding but also specifying a test period immediately prior to the production of the project report.

- Focus on code: due to the limited timespan of the project it was necessary to focus almost exclusively on the production of code rather than documentation, which was limited to that which was included in the report.

5. GAME DESIGN

5.1 Game model

Broadly speaking, the game model as specified in *Section 1.1.1 Aims* could be represented by three main areas of functionality – fief management, army management and household management – joined together by various secondary functions, such as movement (see *Figure 9*). However, a more detailed model needed to be developed before implementation could begin.

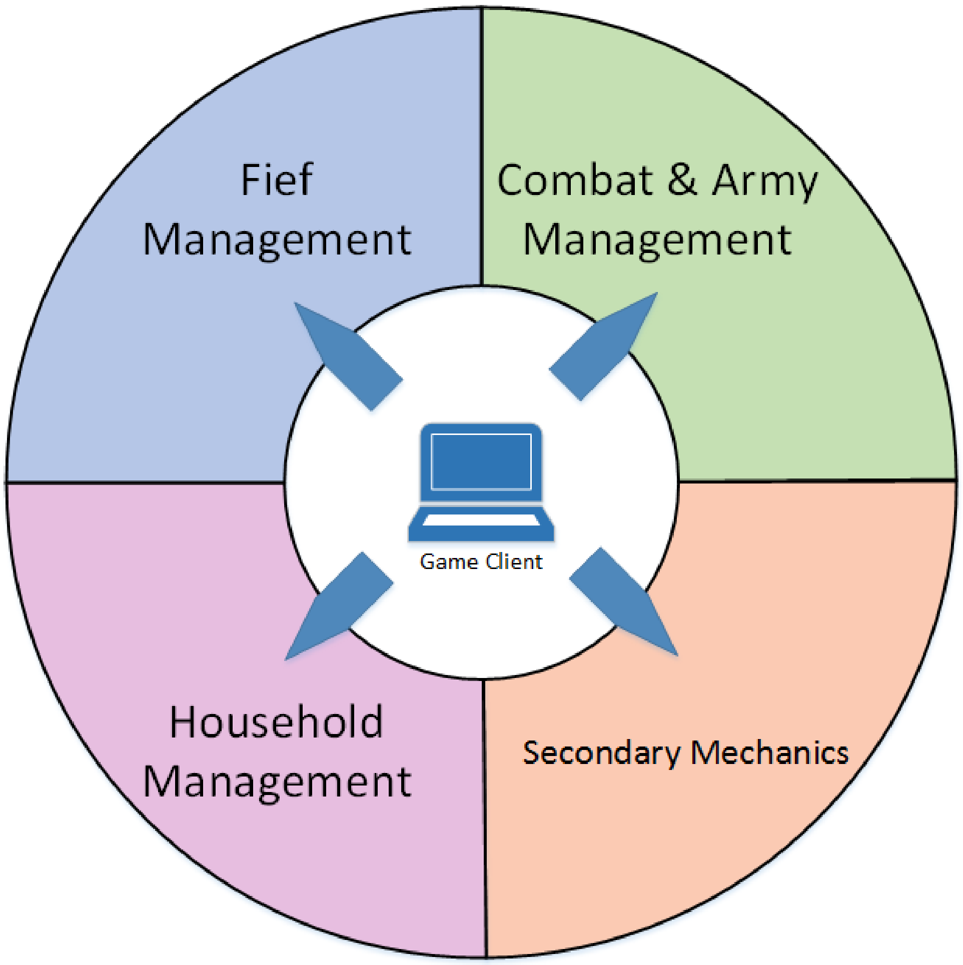


Figure 9: Basic game model

5.1.1 Game setting

The game engine is intended to model the setting and circumstances of Medieval Europe; ultimately, it will allow scenarios to be created for any period during the high to late medieval period (roughly 1100AD – 1450AD). As seen in *Section 2.1.4* of the *Literature review*, much data is available on the medieval era and, crucially, some very useful information on characters and places was publically available on the website of Albert Nofi and James Dunnigan's MMORPG, *The Hundred Years War*, (unfortunately, no longer available) – see Nofi, A.A. and Dunnigan, J.F. (1997).

The decision was taken to make the default period 1194, a time leading up to the drawing up of the Magna Carta and the First Baron's War, in which there were uneasy relationships between England and its bordering countries, Scotland and Wales, potentially leading to events of interest.

Importantly, historical data could be readily sourced for this period from a variant of Nofi and Dunnigan's game for which the data had been archived. The decision to limit the geographical area to that of mainland Britain removed the need for sea transport and ensured that the game world could comfortably be hosted on a single server.

To convey both the nature and setting of the game, it was decided to give it the title '*Overlord: Age of Magna Carta*'. The game engine underlying *Overlord* was named the *JominiEngine*, after the 19th Century strategist and military theorist Baron Antoine-Henri Jomini; it may have perhaps been more appropriate, given the broad scope of the game content (addressing both military and political issues), to name the engine after Jomini's contemporary, Carl von Clausewitz but that name was already being used by another game engine.

5.1.2 Game time model

Broadly speaking, computer games, be they stand-alone or multiplayer, can be divided into two main types as regards the way in which they model the passage of time:

- Turn-based, in which each turn represents a specific period of time (many 'traditional' wargames fall into this category). Progression between turns is handled in one of two ways:
  - Players plan and perform actions sequentially, the game progressing to the next turn when all players have finished.
  - Players plan actions concurrently with each other, and the game engine then performs all actions simultaneously, after which it progresses to the next turn.

- Real-time, in which all players perform actions concurrently and the game progresses at a regular, albeit in many cases accelerated, rate.

It was decided that the *JominiEngine* would be primarily turn-based, but include some real-time aspects; each turn represents a season of 90 days duration (spring, summer, autumn, winter) and a single turn will be processed for each real-world day, an update at midnight allowing the transition between turns. Players are free to login at any time during the day to perform their actions concurrently, with each action taking a specific number of game-world days to perform. Some actions (battle, for example) will have immediate outcomes, whilst the effects for others (adjusting the tax rate for a fief, for example) are calculated during the daily update. Ideally, this allows a combination of real-time cooperation between players, whilst also providing 'time out' to plan; additionally, it gives the players flexibility regarding the way in which they fit the game into their available time.

5.1.3 Game goals

Sicart (2008) posits that a game should have concrete goals, leading to an achievable victory, but that it should also provide a degree of freedom for its players. In addition to this, as Dunnigan (2000) points out, many players of historical games are looking to compare their own performance with that of historical figures, to see whether they can reverse the tides of history.

Accordingly, the decision was made to set the following victory conditions for the game:

Victory conditions	Game type
Total: all fiefs owned by one nationality	Team
Historical: conditions based on a conflict of the period	Team
Score: based on PlayerCharacter's fiefs owned, population controlled and stature (Note: this was later amended to also include finances)	Individual

Table 3: Victory conditions

These provide readily understood and attainable objectives but also allow the players a great deal of freedom to pursue their own agenda (for example, a private grudge match with another player, or the development of an economic empire). Also, the victory conditions need not be mutually exclusive; it would be possible for one team, for example, to win the game based on historical conditions but for the player with the highest individual score to come from the losing team.

5.1.4 Game mechanics

As Crawford (2002) observes, compiling a list of verbs that the developer would want the player to be able to perform in the game can be a useful tool in identifying the underlying mechanics.

By using this technique, the following initial list of verbs was compiled for the game:

Verb	Explanation
Move	Includes PCs, armies, NPCs
View	See other game objects in the vicinity
Marry	Marriage between a PC (or family member) and an NPC from another family
Sire	Get wife pregnant to ensure the continuation of the PC's lineage
Inherit	An NPC family member inherits from a deceased PC, allowing the player to continue in the game
Manage	Manage a fief, household or army
Annex	Take possession of a fief using non-hostile or hostile means
Besiege	Besiege an enemy fief (possibly resulting in annexation)
Battle	Encompasses Attack and Defend
Lead	Lead an army
Pillage	Pillage an enemy fief
Hire	Hire an NPC
Transfer	Includes funds between fiefs or between players, or troops between armies
Bar	Bar an individual or an entire nationality from a fief

Table 4: Main game verbs (actions)

From these verbs a list of core game mechanics were compiled, which could then be translated more readily into object-oriented programming structures and functions. As suggested by Sicart, the core mechanics were sub-divided into primary (those whose employment have a direct influence on the achievement of victory) and secondary (those that do not directly result in the achievement of victory but nevertheless are useful to the player). In addition, some mechanics are defined as compound; that is a number of tertiary mechanics that, when combined, comprise a core mechanic.

Primary mechanics	
Movement	
View armies in fief	
View characters in fief	
Marriage (composite)	
Sire an heir (composite)	
Fief management (composite)	
Recruit army	
Besiege fief (composite)	
Battle	
Secondary mechanics	
Army management (composite)	
Pillage	
Hire NPC	
Bar character or nationality	
Add/remove NPC to/from entourage	
View game information updates	
Transfer funds between players	
Adjust overlord tax rate	
Composite mechanics	
Core mechanic	**Constituent tertiary mechanics**
Fief management	Adjust tax rate; Adjust various fief expenditures (garrison, officials, infrastructure, keep); Transfer funds between treasuries; Appoint a bailiff; View previous/current financial status
Besiege	Negotiate; Reduce defences; Storm keep; Pillage; End siege; Battle (optional)
Army management	Set army standing orders (based on aggression and combat odds); Appoint a leader; Maintain army; View troop numbers; Recruit additional troops; Disband army
Marriage	Propose engagement; Respond to proposal; Process marriage
Sire an heir	Impregnate wife; Process childbirth

Table 5: Game mechanics, derived from verbs

As one of the requirements of *Overlord* (see *Section 1.3.1.1* of the *Requirements and specification*) is that it provides a system administrator role, it was decided to extend the mechanic concerned with information updates to also include the facility to collect metrics and to view error messages, system events, etc. This feature would require the ability to filter messages by type and recipient.

It was also necessary to consider other mechanics, not directly player-initiated but nevertheless necessary for the operation of the JominiEngine; they can be termed 'Housekeeping mechanics' and examples include:

- Fief seasonal update (population growth, industry growth, calculation of status, etc.)
- PC/NPC seasonal update (check health, reset days available)
- Death processing (inheritance, removal from game, etc.)
- Processing of scheduled game events (e.g. birth, marriage)
- Scores update
- Army attrition

As with other aspects of the *JominiEngine*, the need to simplify the initial version had an influence on the choice of mechanics. Accordingly, the following mechanics were considered before being disregarded as not being essential to the core game:

- Assassination
- Kidnap
- Train (to acquire enhanced attributes or traits)
- Joust
- Religious-themed mechanics (for example, excommunication and crusade)
- Raid (a lesser version of pillage)
- Seduction

5.1.5 Game character model

Crawford (2013) spoke of the need for 'personality models' in order to more realistically define interaction between game characters; this particularly refers to NPCs who, unlike PCs, do not have their behaviour directly influenced by the player. This can be done by allocating characters a number of attributes to represent their abilities in various areas; these can then be directly mapped to class attributes when designing the program.

The following attributes were initially decided upon for characters in the *JominiEngine*:

- Stature
- Leadership
- Protection
- Endurance
- Attack
- Management
- Guile
- Virility
- Loyalty
- Maximum health

In order to focus on a few key attributes, it was decided to combine some existing ones into a single composite; for example, leadership, protection, endurance and attack were combined to form the 'Combat' attribute which influences both army leadership and survival in combat situations. The final list was:

- Stature
- Combat
- Management
- Virility
- Maximum health

In addition to attributes, each character was assigned a number of 'traits' which either refines their ability to perform certain mechanics or directly influences particular attributes. A single trait typically embodies a particular trait which influences, positively or negatively, a number of mechanics and attributes; for example, the trait 'Robust' positively influences both Virility and the character's ability to perform the 'NPC hire' mechanic.

In the interest of simplicity and to allow the concept to be more easily tested, it was decided to only create a very limited number of traits and to allocate only 2-3 per character. Additionally, care was taken to ensure an even spread of positive and negative influences. The traits decided upon were:

- Command
- Chivalry
- Abrasiveness
- Accountancy
- Stupidity
- Robust
- Pious

5.1.6 Game resources

Much of the enjoyment in any game is derived from the success with which a player can utilise the, often scarce, resources available in order to outwit their opponents and achieve victory. Management of these resources is fundamental to the game dynamics and the way in which players interact, and thus needs to be carefully designed. It is necessary, therefore, to examine which resources will be provided by the game and address the manner in which they are provided, what they can be used for, and to what extent they can be replenished. In a historically-based game, there is also the added burden of ensuring that resources are depicted in a realistic, albeit abstracted, fashion.

The resources initially identified for *Overlord* were money, population and game days; after further consideration it was felt that NPCs could also be classed as a type of resource.

Resource	**Increased by**	**Reduced by**	**Used for**
Money ‡	Fief income; pillage/siege; transfer between players	Pillage/siege; transfer between players	Recruitment; NPCs; fief expenses; family expenses
Population	Population growth	Pillage/siege	Recruitment; fief productivity
Game days	Passage of time (seasonal update)	Negative skills	Performance of game actions
NPCs	Birth (family); re-spawning (non-family)	Death, hiring by another player	Assuming roles of responsibility
Troops ‡	Recruitment, transfer between players	Combat, transfer between players	Combat

‡ denotes resource is tradable between players

Table 6: Game resources and their characteristics

It was considered important to provide flexibility in the way in which resources can be utilised, thereby requiring the player to strategise; care was also taken to ensure that the means for replenishing resources were historically authentic, given *Overlord*'s level of abstraction.

When designing mechanics, care was taken, where possible, to use formulas that modelled real life processes (see *Appendix B: Formulas*). The exception to this rule was the choice, taken for the sake of simplicity, to re-spawn non-family NPCs in order to provide a steady 'employment pool' from which to recruit.

5.1.7 Game rules

Sicart (2008) defines game rules as the 'possibility space', stipulating which game mechanics are accessible to the player at a given point in the game. Once the rules have been determined, they can be readily mapped to predicates and conditional checks in the game program.

When deciding on rules for the *JominiEngine*, many were implicitly dictated by common sense; for example, the value of the fief tax rate must lie between 0-100, and one army can only attack another if both armies are in same fief. However, it is still necessary to take note of such rules, as they will require conditional checks in the program.

In some cases, in the interests of enjoyable gameplay, decisions have to be taken which may contradict the application of common sense. An example of this is the implementation of the 'Besiege' mechanic; if a played PC is in a keep that is under siege, it would be sensible to ensure that he could not perform the mechanics required to administer his other fiefs, or to order the movement of armies. However, this would so severely limit the player's actions as to possibly make the game unplayable. In the *JominiEngine*, therefore, a player is allowed to perform certain mechanics when under siege (send an army to relieve the siege, etc.), although other actions (for example, moving the PC to another fief) are still prohibited.

Some rules in the *JominiEngine* are dictated by the need to maintain, where possible, historical accuracy. Examples of this include:

- A fief owner cannot grant a fief title to another character if he is the ancestral owner of the fief.

- The practice by medieval nobles of using marriage as a tool to acquire stature is represented in the *JominiEngine* by the rule that marriages can only be arranged between characters that are members of played PCs; NPC employees are simply not of sufficient rank.

- Unemployed NPCs are not to be located in the court of a fief; Courts contain only nobles, their families, and other members of their household.

Other rules have been impacted by the need to simplify the initial version of the *JominiEngine*. For example:

- In the inheritance mechanic, PCs must be male and inheritance is relayed exclusively from father to son.

- It was decided not to model the effects of terrain upon combat, or to map the effects of one troop type upon another.

In some cases, the decision to simplify was made in the interests of gameplay. For example:

- The fief management mechanic entails the adjustment of only a small set of expenditures (garrison, officials, infrastructure, keep), rather than requiring the player to make the myriad calculations that would be required in reality; concepts such as corruption, for example, were considered but ultimately rejected. Whilst reducing complexity, this decision was made primarily to prevent players being buried under a mountain of 'number crunching'.

- Similarly, maintaining an army in the field is a straightforward matter of allocating the appropriate funds, rather than arranging for the purchase and delivery of different types of supplies.

5.2 Game engine

The next stage of the project involved translating the abstract game model into a concrete design for the *JominiEngine*.

5.2.1 Class definition

Four main classes had been identified in the game model – PC, NPC, Fief and Army. Against each class were listed the mechanics in which they would be the primary object involved; this would then be used as a guide when deciding in which class to place the resulting methods.

PC/NPC:
- Movement
- Hire NPC
- Marriage
- Sire
- Add/remove NPC to/from entourage
- Death processing
- Seasonal update

Fief:
- Fief management
- Adjust overlord tax rate
- Pillage
- Besiege
- View armies in fief
- View characters in fief
- Bar character or nationality
- Transfer funds between players
- Seasonal update

Army:
- Recruit
- Army management
- Battle
- Pillage
- Besiege
- Attrition

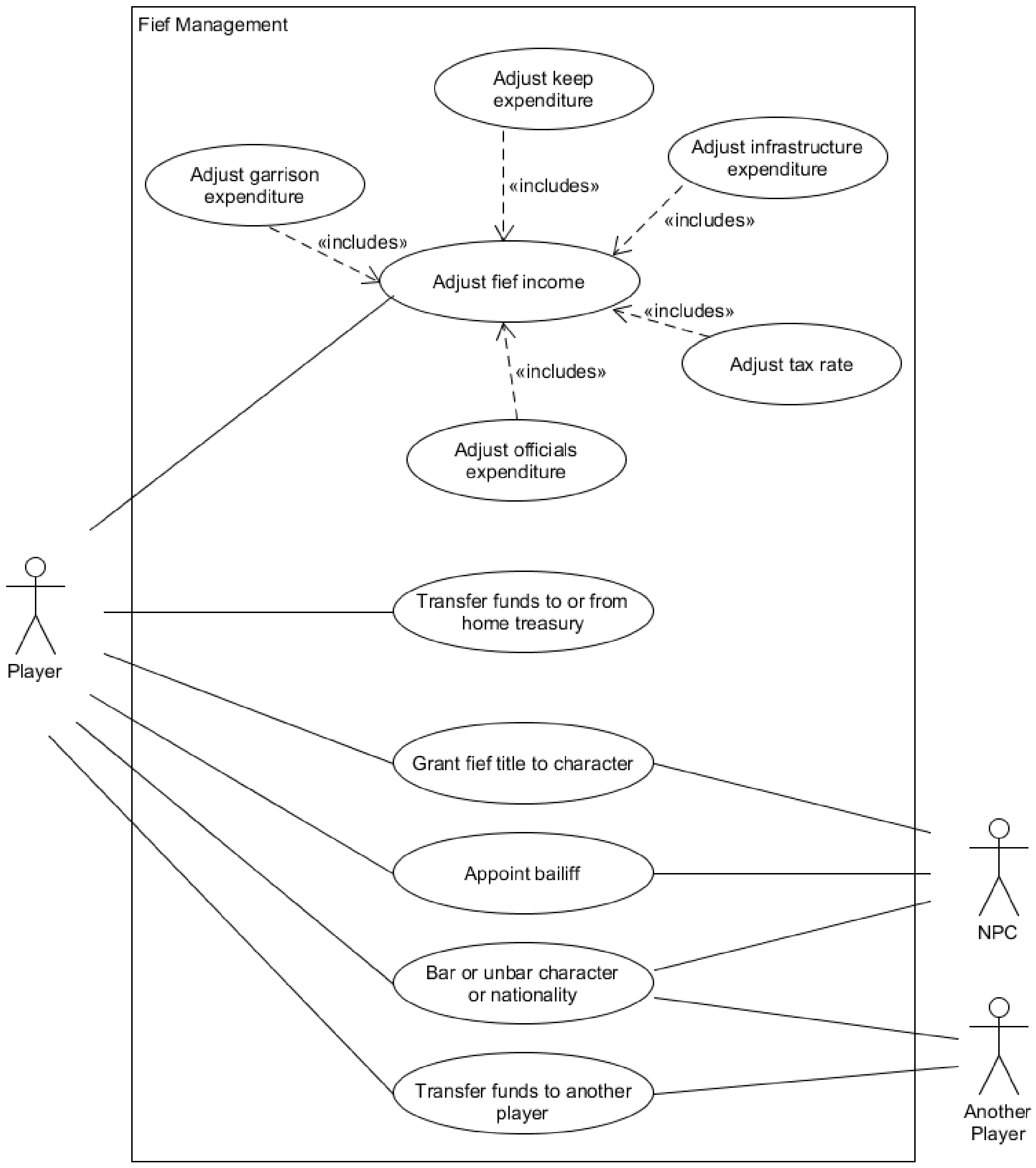

Fief Management
Adjust keep expenditure
Adjust infrastructure expenditure
Adjust garrison expenditure
«includes»
«includes»
«includes»
Adjust fief income
«includes»
Adjust tax rate
«includes»
Adjust officials expenditure
Player
Transfer funds to or from home treasury
Grant fief title to character
Appoint bailiff
NPC
Bar or unbar character or nationality
Transfer funds to another player
Another Player

Figure 10: The main activities involved in fief management.

Potential attributes were now identified for each class that would both define it and allow its participation in each mechanic.

PC and NPC attributes included:

- The 'personality' attributes identified in *Section 5.1.5*.
- Those that defined identity; for example, ID, first name, surname, age, sex, nationality, language.
- Those required for mechanics; for example, spouse, sire (father), head of family, pregnant (all required for marriage and sire), location (required for movement), days (required for many mechanics).
- Those required specifically by PC; for example, purse.
- Those required specifically by NPC; for example, salary

Fief attributes included:

- Those that defined identity; for example, ID, name, province, kingdom/country, language, industry level, field level, population, GDP.
- Those required for mechanics; for example, keep level, numbers of troops by type (required for siege), tax rate, garrison expenditure, infrastructure expenditure, income, bailiff (required for fief management), terrain (required for calculating movement cost).

Army attributes included:

- Those that defined identity; for example, ID, owner.
- Those required for mechanics; for example, numbers of troops by type (required for battle, siege, pillage).

The attributes were then reviewed to see if any were derived values that did not require to be stored and, as a result, several were removed; for example:

- Fief income can be derived from GDP, expenses and tax rate.
- Fief GDP can be derived from the field level, industry level and population.

The class list was then reviewed to identify the requirement for any additional classes. As a result, several new classes were added:

- The similarities between PC and NPC immediately suggested the need for a superclass, Character, containing all common attributes; it would be extended by PC and NPC, which contained only those attributes necessary for their specific needs.

- Character traits are shared amongst a number of objects and their application relies upon several pieces of data; it was decided, therefore, that a separate Trait class would be useful, containing attributes for name and a collection of trait effects.

- All characters contain nationality information, necessary for the bar mechanic and for determining which side is in possession of a fief (traced through the owning PC's nationality). The decision was taken to create a simple Nationality class to hold this information.

- All fiefs contain terrain data, necessary for the calculation movement cost (and potentially for future enhancements to the battle mechanic); as many fiefs would have the same terrain type, it was sensible to create a distinct Terrain class containing description and cost attributes.

- Similarly, all fiefs and characters have a language attribute which is used in the recruit mechanic and influences fief loyalty and bailiff effectiveness, so a Language class was created to hold this information, containing name and dialect attributes.

- In order to implement the movement mechanic, a graph structure would have to be created to map the fiefs into a coherent infrastructure. Utilising the *QuickGraph* library, the HexMapGraph class was created to provide this facility and to allow access to useful methods, such as the calculation of the shortest path between two nodes (fiefs).

- In order to provide a political and administrative structure to the game, all fiefs contain province data, also required for the fief management mechanic. The decision to create a Province class not only permitted a single province to be shared amongst many fiefs but also allowed the overlord and overlord tax rate attributes to be removed from the Fief class. The methods concerned with adjusting the overlord tax rate could also be moved into this class.

- To keep track of game-time (year and season), used for the calculation of character age and for the calculation movement cost (variable, depending on season), it was decided to create a GameClock class, accessible to all game objects (probably through a static class).

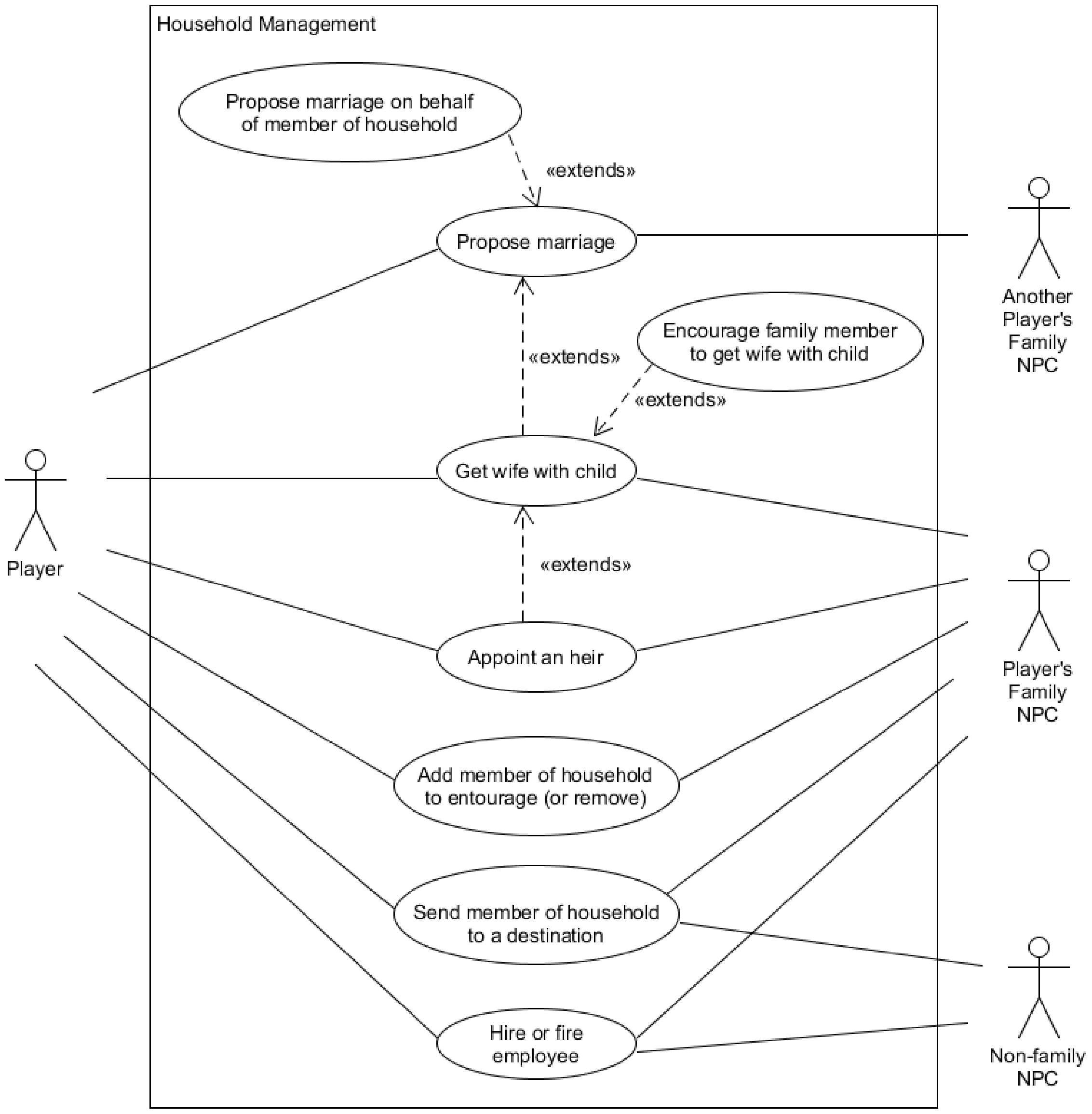

Household Management
Propose marriage on behalf of member of household
«extends»
Propose marriage
«extends»
Encourage family member to get wife with child
«extends»
Get wife with child
«extends»
Appoint an heir
Add member of household to entourage (or remove)
Send member of household to a destination
Hire or fire employee
Player
Another Player's Family NPC
Player's Family NPC
Non-family NPC

Figure 11: The main activities involved in household management.

5.2.2 Methods

Each mechanic identified in the game model was mapped approximately to a method (or collection of methods), the probable location (class) for each method having been decided during the class definition phase (*Section 5.2.1*).

The rules defined in the game model were then used to draw up initial lists of predicates and conditional checks for the main methods. For example, the following checks were identified for a marriage proposal:

- The groom is male
- The bride is female
- Both bride and groom are ‘of age’ (defined as 14 in *Overlord*)
- Neither bride nor groom can be already engaged
- Neither bride nor groom can be already married
- Both bride and groom must come from families of played PCs (also the groom can himself be a played PC)
- Bride and groom must come from different families

This process was not only useful in planning the construction of a method, it often highlighted the need for additional attributes in the class, especially for holding state information. Examples of attributes identified in this way include:

- fiancée (to check for an existing engagement)
- englishBarred (to ensure English characters cannot enter the keep in this fief)

Given that *Overlord* is trying to model real life processes, it was considered important to try to reflect this in the formulas employed in the methods. Consequently, formulas were specified for all important processes, grouped into broad areas covering combat, fief and character. Examples of formulas include:

- For combat, the calculation of: an army leader’s leadership value, the chance of bringing an enemy army to battle, the chance of winning a battle, the effects upon a fief of a pillage, the chance of a leader being injured in battle.

- For fief, the calculation of: population growth, GDP, the effect of tax rate change on fief loyalty, the chance of unrest or rebellion, the effect of bailiff attributes on fief income, the relationship between infrastructure expenditure and the growth of a fief’s fields and industry levels.

- For character, the calculation of: a character’s stature , the way in which traits are applied to game mechanics, an NPC’s ‘worth’ as an employee (i.e. potential salary), the chance of character death during the seasonal update, the extent to which a child’s attributes will vary from that of its parents.

Also examined were other game aspects, such as:

- The time taken (in game days) to perform the various mechanics.
- Given *Overlord*’s time model, at which point specific object attributes should be updated. For example, battle casualties occur immediately, as a part of the battle mechanic, whereas population growth occurs during the daily seasonal update.

This process was very useful in identifying those mechanics that would be better broken down into a number of ‘sub-methods’ in order to provide discrete access to component data. This is especially important for components that are used in more than one mechanic. For example, when calculating fief loyalty, it is necessary to into account the following:

- Tax rate
- Surplus
- Bailiff attributes (including language)
- Bailiff traits

- Officials expenditure
- Garrison expenditure

Once the core objects and methods have been identified, it is often helpful to view them in diagrammatical form, allowing relationships between objects and mechanisms to be more clearly defined, whilst also helping with the process of designing the test client.

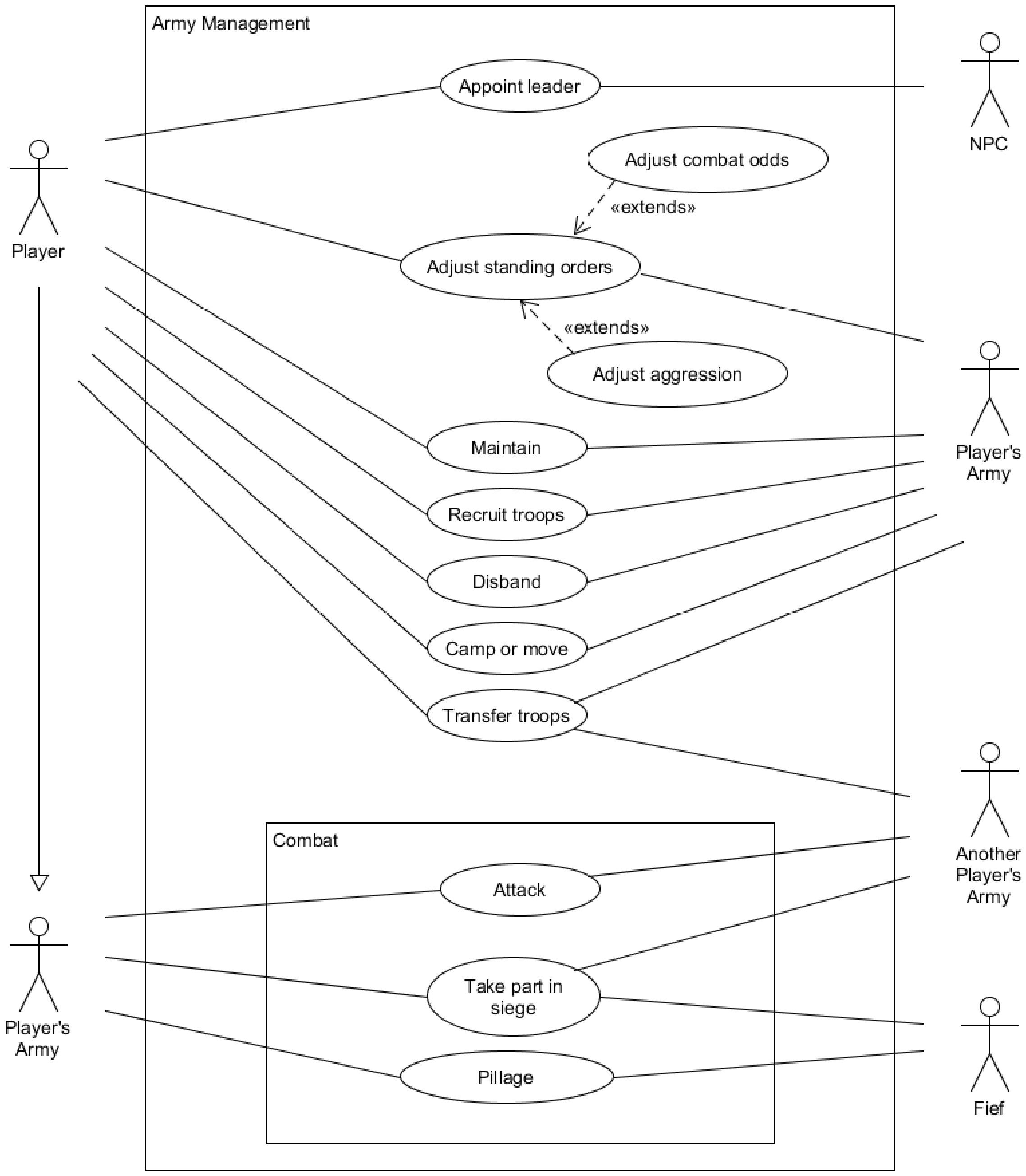


Figure 12: The main activities involved in army management and combat

5.2.3 User Interface

Although *Section 1.3.1.2* of the *Requirements and specification* only stipulates the creation of a test client with a minimal user interface, it was still necessary to consider in some detail how best to present the functions and information that the player would require in order to effectively play the game.

Regarding the nature of the interface, there were two main choices – i) text-based, presenting a series of text menus and requiring user input at the command line; ii) graphical, making use of elements such as buttons and drop-down menus. The graphical option, readily available through the *.NET System.Windows.Forms* namespace, was felt to be the most appropriate option, being both easier to use and allowing the more efficient presentation of interface components. It was also felt that it would more easily facilitate the creation of a fully realised hexagon map, an optional requirement of the client interface.

The game objects and mechanics were assessed with a view to creating logical groups of functions that could be accessed from a single point in the interface. It was also useful at this point to examine the online help pages of Nofi and Dunnigan's *The Hundred Years War* to see how its interface had been organised, although allowances had to be made due to its text-based approach.

The following initial functional areas were identified:

- PlayerCharacter information
- Movement
- Fief management
- Army management and combat
- Household management
- In-game messages (i.e. information on game events)
- King functions
- System administrator functions

The process of considering what information needed to be available to the player at certain points in the interface, also led to an appraisal of the most efficient way to retrieve the necessary data and, therefore, the most appropriate structure for its storage. The two main decisions that needed to be made were:

- Should classes contain local collections of objects for faster retrieval, rather than relying on the filtering of all objects of that type. For example, should the Fief class contain a collection of characters located in that fief, or should all character objects be filtered by their location attribute? If the former, then could the Character class location attribute be removed entirely?

- Should certain attributes consist of embedded objects, again for faster access to the object data, or should they simply hold the ID of the object, which could then be retrieved from a master collection? For example, should the Fief class owner attribute consist of a PlayerCharacter object or merely the object ID?

In most cases, the decision made was to hold collections of objects (or object IDs) within the class, rather than having to filter all objects. For example, the PlayerCharacter class would hold a collection of associated NPCs (family members and employees). However, it was also decided to take a 'multiple redundancy' approach by retaining the associated attributes within those classes. Using the example above, NonPlayerCharacter class would retain the familyID attribute. This would allow flexibility in the retrieval of data, depending on the particular situation.

Decisions on the type of attribute to use (object or ID), were made on a case-by-case basis, depending on the perceived frequency with which they would be accessed. For example, it was decided that the PlayerCharacter class NPCs collection would be frequently accessed and should, therefore, contain Character objects. Conversely, it was decided that the Character class fiancée attribute would be infrequently accessed and would, therefore, contain a character ID, rather than the actual object. In most cases, the attribute was initially left as an ID until access frequency could be verified.

Unexpectedly, the decision to employ a graphical user interface also effected the choice of IDE used in the project; for reasons of performance in Linux environments, the *MonoDevelop* and *Xamarin Studio* IDEs make use of GTK# to create graphical interfaces, rather than Windows.Forms (although the *Mono* environment does support the execution of Windows.Forms applications). As the project developer had no previous experience of using GTK#, it was decided to proceed with the use of Windows.Forms which, in turn, meant that *Microsoft Visual Studio 2010* (and by extension Windows) would need to become the primary development environment. To ensure cross-platform functionality, the application would be regularly tested in Linux, using *MonoDevelop* to debug any identified errors.

5.2.4 Architecture

Although the first stage of the project plan (see *Section 4.1*) called for the prototype to be developed using locally stored data (i.e. not in a client-server configuration), it was still necessary to consider some issues of architecture in order to more easily facilitate its later introduction.

It was decided to create collections holding specific object types (PC, NPC, Fief, etc.); these collections would use the Dictionary class, allowing the reliable retrieval of an object using its ID. These collections would also be used during the daily seasonal update, allowing easy iteration through objects of a particular type. Each collection would be created at the appropriate point in the implementation phase (i.e. when that object type was first required by the application).

In order to allow all game objects to have access to all other objects, it was decided that these collections would be initially contained within the Form1 class, which would instantiate all other game objects.

It was decided that the Form1 class would also need to contain attributes used to keep a track of which fief, character, army, etc. was currently being viewed in the interface and which PC objects had been designated to fill the game roles of king, herald and system administrator. These attributes would allow the corresponding objects to be easily passed to methods as parameters.

Looking ahead to the introduction of a client-server architecture, thought was given to those attributes that would need to be accessible to all game clients (for example, the king, herald, and sysadmin attributes; the object collections) and those that were required only by that particular client (for example, which PlayerCharacter was being played, which fief was currently being viewed). It was thought that, at a future point, a static class (or classes) might be used to make these attributes easily accessible but, as mentioned above, they would be stored in the Form1 class initially.

Thought was also given as to the later-stage architecture with regard to ensuring consistent communication of game state and information updates (for example, 'news' items about battles, sieges, marriages, deaths) between the server and clients. The adoption of the MVC or observer pattern was thought appropriate as it allows a semi-automated approach to communication and, as a result, the decision was taken to create the appropriate interfaces, although they would not be properly employed until later in the implementation.

6. IMPLEMENTATION

Implementation mainly proceeded based on functional modules, although, due to their interconnected nature, it was frequently necessary to revisit modules in order to implement new features (see *Appendix C: Overview of files and content*). In general, however, implementation progressed in the following phases:

- Initial creation of objects and basic user interface
- The fief: its management and its place in the game world
- Movement, travel interaction and hiring
- Backend database management system
- Household management, inheritance and in-game information messaging
- Army management
- Combat
- Royal functions and overlord functions
- Victory conditions and calculation
- System administrator functions

In addition, the developer planned periods towards end of implementation to allow for:

- Code refactoring
- Testing

See *Figure 13*, *Appendix A: Diagrams*, for a UML entity relationship diagram showing the relationships between the main *JominiEngine* classes.

Certain decisions were taken during the design phase (see *Section 5.2*) or early on in the implementation phase regarding coding practices, although they were subject to change due to emerging circumstances. These decisions included:

- The use of collections ('master lists') of particular object types that would allow efficient and reliable retrieval of specific objects for use in methods, and the straightforward iteration through a collection, which was very useful for the daily seasonal update.

- The Dictionary was chosen as the type of collection used for the master lists as it allowed the fast retrieval of an object using its ID. One drawback of using the Dictionary was the difficulty in retrieving a collection member by its index position but the need for this was infrequent.

- In order to give a global scope to the master lists and, subsequently, other attributes and utility methods, it was decided to make use of static classes for the initial version of the engine, mainly in the interests of simplicity. It was recognised, however, that it would probably be more effective to switch to an alternative method (such as the Singleton pattern) for future versions to facilitate, for example, the creation of user-defined extension methods (something that cannot be achieved with a static method). Later in the implementation, distinct static classes were used to separate attributes associated with the client and server side of the JominiEngine.

- In the early stages of implementation, error reporting and user feedback was provided exclusively through the use of the MessageBox class (System.Windows.Forms). It was later decided that error reporting should make use of the exception handling in those circumstances where the error was predictable and was likely to affect the execution of the game engine (for example, in a constructor, to alert the user to the inability to create an object) or where exceptions provided the simplest way to check for incorrect user input (for example, inputting a string instead of a integer). Care was also taken, where possible, to carry out conditional checks to ensure that errors should not arise (see *Section 5.2.2*).

6.1 Initial implementation

The initial phase consisted of the creation of the main game classes specified in *Section 1.3.1.1* (Character, PC, NPC, Fief and Army), the HexMapGraph class, which used the *QuickGraph* library, and the basic user interface (an instance of Windows.Form). Other objects, identified during the design phase as being required, were also created (Trait, Language, and Province).

The HexMapGraph class is employed essentially to tie together the fief objects into a coherent and navigable game world and its implementation at a very early stage was, therefore, vital. It consists of two main elements:

- A collection of vertices, which are the nodes in the graph (i.e. the components that you are wishing to connect). In this case, a vertex maps to a Fief object.
- A collection of edges, constructs used to establish a connection between two vertices (the source and target). The type of edge chosen for HexMapGraph was the TaggedEdge, which allows an edge to be identified and retrieved, based on its tag; in this case, the tags mapped to the direction of the connection on the hexagon map (‘W’, ‘NW’, ‘NE’, ‘E’, ‘SE’, ‘SW’).

A third element was subsequently added:

- A Dictionary associating each edge with a double value, representing the travel cost. This was used to facilitate a *QuickGraph* method that used Dijkstra's algorithm to identify the shortest path between two vertices.

The most important function of HexMapGraph is to return a fief object which can then be used as a character’s next location (or, in some cases, a collection of fiefs representing a path along which the character will sequentially travel). The *GetFief* method is used to identify and return a fief, using the source (current fief) and direction (string) passed in as parameters. The key segment of code that achieves this is the following (where ‘f’ is the source fief, ‘direction’ is the desired direction of travel, and ‘myFief’ is the fief to be returned):

```
// iterate through graph edges
foreach (var e in this.myMap.Edges)
{
    // if source matches f, check tag
    if (e.Source == f)
    {
        // if tag matches, get target
        if (e.Tag.Equals(direction))
        {
            myFief = e.Target;
            break;
        }
    }
}
```

At this stage, the user interface was very basic, consisting of a single container which displayed the text output from a test method; this method allowed the developer to test that object attributes could be correctly displayed and that the character could move from one fief to another on a simple (seven fief) map. As each functional module was developed, the user interface was adapted accordingly, usually requiring the addition of a container, to house information displays and objects, and a new drop-down menu item to provide access to the new container.

For the sake of convenience, a separate form, SelectionForm, is also used in a number of places to allow the listing and selection of game objects for a specified purpose (for example, candidates for a bailiff appointment, or armies in a fief).

6.2 The fief: its management and its place in the game world

As can readily be seen from *Table 6* (*Section 5.1.6*), the Fief class is integral to the production of two main resources, money and population, both of which are factored into a player's individual score; indeed the ownership and shrewd management of fiefs can be identified as the single most important component in the attainment of victory. It was decided, therefore, to make fief management the first major module to be implemented.

6.2.1 Fief management

Figure 10 (*Section 5.2.1*) shows fief management to comprise of a small set of relatively simple operations and components that interact in a quite complex manner. It was felt important, therefore, that complex procedures should be broken down into separate methods, thus allowing easier identification and testing.

Fief management methods can be grouped into three main categories, most of which are located in the Fief class (but some of which are in the Form1 or SelectionForm classes):

- Those directly used by the player, via the user interface, to: i) adjust the tax rate and the four expenditures (garrison, officials, infrastructure and keep); ii) transfer funds between the player's home fief and his other fiefs; iii) appoint a bailiff; iv) grant the fief title to an NPC. Additionally, although not called for in the original design, there is a set of extension methods to detect and adjust overspending (i.e. when insufficient funds are available in the fief's treasury).

- Those used to calculate changes to key attributes such as the fields and industry levels (both dependant on infrastructure expenditure), keep level (dependant on keep expenditure) and loyalty (dependant mainly on changes to the tax rate and fief surplus but also on garrison and officials expenditure). These methods are mainly used during the seasonal update – *UpdateFief* – but also to display information to the player.

- Those used to calculate derived data such as GDP (based on fields and industry levels) and income (based on GDP, and tax rate) and surplus (based on income, expenses, overlord tax rate, and fief status). In addition, many of these elements (loyalty, income, expenses) are effected by the attributes of the fief bailiff, but only if the bailiff has spent 30 days in the fief during that season. Again, these methods are used during the seasonal update or for information displays.

As work proceeded on the fief management interface, the need became clear for better way to display the financial data required by the player in order to make sound decisions; information needed to be available for the previous and current seasons, and also an estimated projection for the next season, based on the player's tax and spending decisions. A consequence of this was the decision to create two new attributes (double arrays) holding a variety of financial data for the previous and current seasons; these would be updated during the seasonal update.

Family expenses are one of the primary expenses involved with fief management; these are basically a combination of employee wages and family member allowances. For the sake of simplicity, it was decided that bailiff expenses would be applied in the fiefs for which they are bailiff and all other expenses (including non-bailiff employee wages) would be applied at the home fief. Family member allowances are derived by using the NonPlayerCharacter *CalcFamilyAllowance* method which basically specifies an allowance based on family function. Additionally, family expenses in the home fief are influenced by the attributes of the head of family or his wife, rather than the bailiff.

See *Figure 39* (*Appendix A: Diagrams*) for an entity relationship diagram showing the influence of fief components upon each other.

6.2.2 The fief's place in the game world

To allow the more accurate modelling of the game world and its political aspects the Kingdom and Rank objects were implemented. This allowed a fief to be associated, via its Province, with its rightful Kingdom and King (the owner of the Kingdom), and also made it possible to identify a fief that was 'under occupation'. Rank allowed stature value to be associated with each place, which was then used when calculating a character's base stature. The Place object was later implemented as a superclass for Fief, Kingdom and Province.

Much of the political landscape of medieval Europe centred on the ancestral ownership of land; it was decided, therefore, to model this in *Overlord*. The Fief class contains the *ancestralOwner* attribute, specifying a PlayerCharacter. Ancestral ownership effects gameplay in a number of ways:

- A PC cannot grant a fief title to another character if he is the ancestral owner; the exception to this is the king.
- If ownership of a fief changes as the result of a siege, fief loyalty will decrease if the ancestral owner has been ousted, but will increase if the new owner is the ancestral owner.
- During an attempt to quell a rebellion in a fief, the chance of success will be increased if the owner of the quelling army is the ancestral owner and the current fief owner is not.

It is worth noting that, during the design phase, the decision was taken to simplify two aspects connected with the fief:

- The handling of money; rather than having to physically transport money around the kingdom, any money generated by a fief is stored in its own treasury and can be transferred into the player's home fief treasury which is then available for the player to spend. Money can also be transferred in the other direction, to support expenditure in a particular fief. The home fief is identified by a PlayerCharacter attribute containing its ID. Note that the PlayerCharacter does have a purse attribute, but only to support future changes to functionality in this area.

- The handling of troops; rather than having Fief attributes containing numbers of specific troop types, any requirement for calling up troops (for example, recruitment or the formation of a militia in the event of pillage) uses recruitment ratios (stored in global variables) to generate the appropriate forces, based on the fief's population. This method is also used, in conjunction with the fief's garrison spend, to generate the garrison in the event of a siege.

6.3 Movement and travel interaction

6.3.1 Movement

The next core module to be implemented was movement. Three movement mechanics were identified as being of use to the player:

- Simple fief-to-fief movement.
- 'Exact route' movement, in which the player specifies a series of directions in which he wishes to travel, starting at the current fief (for example, 'w,nw,e,nw').
- 'Go here' movement, in which the player gives the fief ID for his destination and is moved there by the least expensive path.

Fief-to-fief movement was achieved quite simply by accessing the *GetFief* method (see *Section 6.1*), having first determined the cost of the move and checked to see if the PlayerCharacter had enough days left to accommodate it. Once the fief has been retrieved, the necessary steps are taken in a separate method (*Character.MoveCharacter*) to remove the PC from the current fief and place him in the target; this involves changing the PC's location attribute to the new fief, removing the PC from the fief's *charactersInFief* attribute (a List of Characters) and adding him to the target fief's *charactersInFief*.

'Exact route' movement required the creation of a new Character attribute, *goTo*, a Queue containing the fiefs to which the character will move. A Queue was chosen as it is specifically designed to sequentially process its collection in the order in which they are added and has methods to look at (*Peek*) or remove (*Dequeue*) the first item in the collection.

Once this attribute was in place, it was a relatively straightforward task to iterate through an array of directions input by the player, retrieving the Fief by using *GetFief* and adding it to the *goTo* attribute. If an incorrect direction entry was encountered, the path would be stopped at the last valid entry. Once the *goTo* has been populated, each of the movement steps is processed in turn until the *goTo* queue is empty or the character's days have been exhausted.

'Go here' movement also makes use of *goTo* to store a series of fiefs retrieved from *QuickGraph's ShortestPathsDijkstra* method.

Given that both 'Exact route' and 'Go here' movement may result in *goTo* queues that the character cannot complete in a single season, it was necessary, during the season update, to check each character's *goTo* queue and process any remaining moves.

In order to more efficiently store terrain data, needed to calculate movement costs, a Terrain class was created and replaced the simple string terrain attribute in the Fief class. Movement costs were initially based on the terrain cost of the target fief, but it was decided that a more realistic way to calculate the cost of a move would be an average of both the source and target fief's terrain costs; this would remove the occurrence of two characters paying different costs for the same move simply because they were going in different directions.

The decision was also taken, during the planning phase, to allow for certain circumstances that might have an effect on travel costs. Therefore, the movement method calls two other methods that calculate a season travel modifier (for example, the cost of travel in winter is twice that of summer) and an army travel modifier, based on the size of an accompanying army.

Another factor to take into account when performing movement is the synchronisation of a PlayerCharacter's entourage. This is done by iterating through the PC's *myNPCs* attribute (a list of Characters) and identifying those that have their *inEntourage* attribute set to true. Rather than performing the full *MoveCharacter* method on each NPC, however, the PC's *location* attribute is used to make the necessary changes and the NPC days are then synchronised with the PC's. Conversely, if an NPC who is in the PC's entourage is given an independent movement order, it is necessary to remove them before processing the move.

It was also decided that, in order to introduce a dynamic element into *Overlord*, unemployed NPCs should move about the map by performing a single random move per seasonal update. This is done by calling the *ChooseRandomHex* method in HexMapGraph and passing in the current fief; it then iterates through the edges collection, identifies all edges with the supplied fief as a source, selects a random edge from this list and returns the target fief.

6.3.2 Travel interaction and hiring

This area concerns the interaction between the PlayerCharacter and the other characters that he encounters in his travels. In the design phase, the decision was made to provide three 'meeting places' for the PC to interact in this way:

- The court of the fief, which provides access to any characters currently inside the keep; for example, the fief owner and his family and employees. A PC may not be able to access the court due to barring.
- The tavern, which is outside the keep and provides access to any unemployed NPCs.
- 'Outside the keep', which is basically the inverse of the court, listing all characters not in the keep.

The need for simplification resulted in a reduction of the mechanics available in the meeting places, although the court may eventually provide access to mechanics such as tournament combat and seduction.

In the initial game engine, the most important activity accessible through the meeting place is the hiring mechanic, which allows a player to examine and hire unemployed (and employed) NPCs to perform duties for his household, chiefly fief bailiff and army leader. The hiring mechanic uses a fairly complex formula to assess the likely worth (i.e. potential salary) of an NPC and then applies a random element to model the negotiation that may occur during the process.

An NPC's potential salary takes into account (including traits) their worth as a bailiff, their worth as an army leader, and their current salary. The resulting salary is then influenced (i.e. possibly reduced) by the hiring NPC's stature, and also by the current employer's stature (i.e. possibly increased). A 'scope of negotiation' is then calculated (between 90% and 110% of the potential salary), against which the hiring PC's offer is compared; a random double, itself influenced by the hiring PCs 'npcHire' trait, is used to ascertain whether the negotiation has been successful or not. Additionally, the NPC's *lastOffer* attribute is used to ensure that subsequent offers from PCs must be for an increased amount.

6.4 Backend database management system

At this stage, it was decided to start to address another of the project's primary objectives (see *Section 1.1.2*), that of scalability. A necessary component of any MMO architecture (there are numerous examples in the *Literature review*, *Section 2.2.5*) is the use of an efficient and reliable backend database management system (DBMS) which can be used to store object sets for the creation of new game instances and also the current game state, enabling an existing game to be re-initialised, should the need arise.

Although inherently reliable, one of the key drawbacks of the traditional relational DBMS is that they are designed for large throughput operations, rather than the small but frequent operations associated with MMO gaming. As mentioned in the Methodology (*Section 3.1.3*), the non-relational DBMS, *Riak*, was chosen for the project due to its purported ability to support a combination of 'big data' and frequent access operations. It was intended to test its suitability, not only in regard to its non-standard design but also its distributed architecture, which might be employed to provide speedy access to a client-base potentially dispersed across a wide geographical area.

The anticipated need for object serialisation in connection with DBMS storage and retrieval meant that it was advisable to address this issue earlier on in the implementation, when the number of object types was still relatively few.

Unfortunately, Riak can only be installed on a Linux platform which, although not a problem as such (the server side of the game engine was always intended to be hosted on a Linux machine), did entail a temporary move to the MonoDevelop IDE and, from this point on, frequent switching between Windows and Linux for testing purposes.

The *CorrugatedIron* library was easily installed and setup to provide the functionality to link to *Riak*, and initial tests indicated that the connection to *Riak* was fast and reliable, albeit with small amounts of data. *Riak* data is organised in a relatively flat fashion which made it difficult to store and retrieve objects of a specific type; this necessitated the creation of Lists containing object IDs which could then be used to retrieve object collections for insertion into the game master lists.

It became obvious at an early stage that the combination of decisions to, firstly, nest objects within other objects and, secondly, to adopt a 'multiple redundancy' with regards to the access of object data, would increase the amount of work necessary for serialisation. *CorrugatedIron* uses the *JSON.NET* library to automatically serialise objects prior to storage but the recursive nature of some of the object relationships meant that additional work would have to be carried out prior to this. For example, the Character *location* attribute contains a Fief object, which itself contains a number of Character objects (in its *charactersInFief* attribute), each of which will contain the Fief object and so on.

It was necessary, therefore, using a combination of constructor and other methods, to substitute object IDs for the actual objects both prior to storage and subsequent to retrieval. Anticipating an impact on *Riak* transaction speeds, the decision was taken to change some existing class attributes from object to object ID, whilst leaving those that would be most frequently used.

The only other issue that arose with the Riak implementation was in regard to the HexMapGraph class which used *QuickGraph* object structures not known by *JSON.NET*. It would have been possible to define JSON schemas specifically for this purpose but it was found to be unnecessary due to the discovery of a *QuickGraph* method that enabled the creation of a graph by passing in a collection of edges. It was only required, therefore, to serialise the HexMapGraph edges collection, substituting Fiefs with Fief IDs on storage and vice versa on retrieval.

When implementing the DBMS functionality, care was taken to ensure that there was a clean interface between the methods referencing *CorrugatedIron* and the rest of the game engine, via the *DatabaseRead* and *DatabaseWrite* methods.

A later extension to the DBMS interface was the addition of functionality (Form1 *ImportFromCSV* and related methods) that allowed the direct import of game data from CSV files into the database, which could then be used to initiate a game.

6.5 Project extension

At this point in the implementation, it became apparent that the deadline for producing the JominiEngine prototype (Friday 23rd May 2014) was not going to be achieved. The main reasons for this were:

- Firstly, the project was more complex than had originally been appreciated; although steps had been taken during the design phase to reduce the complexity of the core engine, it nevertheless involved the intricate interaction of large objects composed of many attributes (Fief, for example, has over 50 attributes if financial data is included). The introduction of each additional functional area entailed a significant increase in complexity, often requiring refactoring of existing mechanics.

- Secondly, given the above, the project was attempting to address too many issues; it became clear that investigation of the architectural aspects of the game engine should be dealt with separately from the core game functionality.

- Thirdly, developer's personal circumstances, specifically the fact that he was in full time employment, were having a negative impact upon the amount of time that could be dedicated to working on the project.

The decisions were taken, therefore, to both extend the deadline for the project to Friday 19th December 2014, and to revise the aims and objectives. The newly agreed aims and objectives were specifically focussed on the production of a core game model, a robust core game engine, and a minimal user interface, sufficient for testing purposes.

6.6 Household management, inheritance and in-game information messaging

The next component to be implemented was household management; this part of the interface provides the facility to view a list of family member and employees (stored in the PC's *myNPCs* attribute), to examine their details, to add them to the PC's entourage, and to give them movement instructions (this is done using duplicate controls to those in the travel interface).

Care was taken to ensure that the player has enough information about NPCs upon which to base their employment decisions, including the display of current responsibilities. Initially, the NonPlayerCharacter *getFunction* method produced a string detailing the NPCs family position (for example, 'Son') and their employment role (for example, 'Bailiff of …'). Later, however, the decision was taken to make a distinction between the role within the family and any employment responsibilities and to provide a new method (*GetResponsibilities*) to derive the latter.

One of the key functional areas identified during the design phase (see *Section 5.1.4 Game model design*) concerned the continuation of the player's family line through childbirth and inheritance. Simply put, in the initial version of the *JominiEngine*, if the player does not ensure that his PC has a son to inherit his mantle, the game will end upon the death of the PC. To achieve this aim, the player can make use of two main mechanics, childbirth and marriage, both of which rely on the use of the in-game information messaging system.

6.6.1 In-game information messaging

The in-game information messaging system in *Overlord* provides several functions:

- The communication of information messages regarding game events. This facility can be extended to include any type of message; for example, some system errors produce a JournalEntry for the system administrator containing relevant details.
- The communication of marriage proposals and their replies.
- The scheduling of certain game events such as birth and marriage.

To facilitate this functionality, two classes were created:

- Journal, which provides a medium in which to store messages. It contains methods to search and retrieve messages based on specified criteria.
- JournalEntry, which comprises the messages itself and contains attributes to store details such as the type of event, personae involved, the location, etc. It contains methods to, for example, check for player interest in a message, and assign a priority.

Although the client-server architecture of the *JominiEngine* was not implemented in this project, it was decided to make use of distinct static classes to model this behaviour, storing client-side data in Globals_Client (for example, the character currently being viewed), game-specific server-side data in Globals_Game (for example, the object master lists), and game-independent server-side data in Globals_Sever (for example, the next available game ID). It was then possible to implement the Observer pattern for the purpose of communicating in-game messages.

Journal objects employed in the initial version of *Overlord* are:

- The *pastEvents* journal in Globals_Client, which contains messages of specific interest to the player.
- The *pastEvents* journal in Globals_Game, which contains all messages communicated during the course of the game.
- The *scheduledEvents* journal in Globals_Game, which contains events that are scheduled to occur at some point in the future (currently just births and marriages). This journal is checked during each seasonal update to see if any events are due to be processed during that update; if so, the appropriate method is called.

An attribute in Globals_Game (*jEntryPriorities*), comprises a Dictionary that associates JournalEntry type and character ID with a priority level; this is used to assign priorities to certain types of messages.

A simple user interface was developed that allowed the player to browse through messages in Globals_Client *pastEvents* and to respond to any marriage proposals in their role as head of the family.

As many in-game events require the creation of journal entries, implementation of the in-game information messaging system was occurred throughout the project, with additions to *jEntryPriorities* being added as and when required.

See *Figure 40*, *Appendix A: Diagrams* for a sequence diagram showing the communication between objects for in-game information messaging.

6.6.2 Childbirth

The childbirth mechanic, mirroring real life, comprises of two distinct steps: pregnancy, which uses the Character *GetSpousePregnant* method, and birth, which uses Form1 *GiveBirth* and related methods.

The pregnancy formula takes into account the wife's age and an average of the couple's virility to arrive at a percentage chance, which is then compared with a random double. If successful, a journalEntry is created and added to the Globals_Game *scheduledEvents* journal, to be processed three updates hence.

The birth process involves a fairly complicated sequence of methods, orchestrated by *GiveBirth*.

- The parent characters are retrieved using the IDs contained in the birth message in *scheduledEvents*, and a new NPC is generated.
- This NPC will derive its attributes from the parents, including amended 'personality' attributes, which could be better or worse than the parents.
- A check is then performed to see if mother and/or baby have died during the process. Although, the Character *CheckDeath* method is used for other occasions, it was necessary to adapt it to account for the enhanced danger involved in the birth process during these times.
- An appropriate message is sent to Globals_Game *pastEvents* (and propagated to Globals_Client *pastEvents*).

If successful, the birth will result in the new NPC being added to the myNPCs attribute of the relevant head of family.

6.6.3 Marriage

Like childbirth, marriage consists of two steps: proposal, which uses Form1 *ProposeMarriage* and *ReplyToProposal*, and marriage, which uses Form1 *ProcessMarriage*.

Proposal is a relatively straightforward process:

- Various conditional checks are performed (see *Section 5.2.2 Game engine design*).
- A proposal message is created for the intended bride's head of family.
- Once alerted to the proposal, the 'accept/reject' controls are enabled in the head of family's journal interface which call *ReplyToProposal*.
- This creates a new message containing the reply, and the original message is also amended to show the reply.
- If the proposal is accepted, a slightly amended version of the reply message is added to *scheduledEvents*, to be processed during the next season update.
- The bride and groom's fiancée attributes are updated accordingly.

The *ProcessMarriage* method simply amends a number of the bride's and groom's attributes (for example, *fiancée*, *spouse*, *familyName*) and the bride is transferred into the groom's family, taking the groom's *familyID* and being transferred from one head of family's *myNPCs* to the other's.

6.6.4 Death and inheritance

The other key process involved in the inheritance area is the 'housekeeping' mechanics for checking and processing a character's death, primarily performed by Character *CheckForDeath* and *ProcessDeath*, respectively. Death in *Overlord* can occur through a variety of circumstances, including battlefield injury, childbirth (mother or new infant), and natural causes.

In order to more accurately model the effects of injury and disease, the Ailment class was created; it contains attributes recording the current effect of the ailment and the minimum effect (some ailments never completely disappear). Ailment processing (during the season update) will incrementally reduce each ailment effect until the finally reach the minimum level, and will remove any ailments that have an effect of 0.

Each character's health is derived from their *maxHealth* attribute, above which their health cannot rise, modified by age and by any ailments they might have. During the season update, *CheckForDeath* performs a check on each character, generating a chance of death based on their health, sex (women are less likely to die), and modified by their 'death' trait.

Death processing is a complex but easily understood process, which basically involves 'tidying up' the character's affairs and then transferring titles, property and other possessions (for example, family members and employees) to the designated recipient PC. In the cause of historical authenticity, it was decided that if a PC without an heir were to die, his king will inherit.

The process involves:

- Removing the character from any positions of responsibility.
- Cancelling scheduled marriages and births, where necessary.
- Removing from their location (Fief object).
- Removing from their spouse (updating the spouse's *spouse* attribute).
- Re-assigning titles, where necessary (these may be inherited by the heir).
- If the deceased was a non-family NPC, they will be re-spawned in a random fief of the same language, using a Character constructor that will vary their 'personality' attributes slightly.

If the deceased was a PC, inheritance varies slightly depending on whether there is an heir:

- If there is an heir, this NPC will be promoted, using a PlayerCharacter and Character constructors tailored for the purpose. The new PC will then inherit his father's properties, positions, titles, armies, sieges, and NPCs, and the *familyID* or *employer* attributes of all of his NPCs will be changed to his ID.
- A similar process occurs if the king inherits with the exceptions that any armies are disbanded, sieges ended, and family members cast out (employees are retained).

See *Figure 38, Appendix A: Diagrams* for an activity diagram showing the main steps involved in death and inheritance.

6.7 Army management

Like fief management, army management is another compound mechanic, although a less intricate one, dealing primarily with less complex classes (Army and Siege).

Recruitment has to be carried out by the player's PC in his current fief and is processed using the PlayerCharacter *RecruitTroops* method. Prior to recruitment, various conditional checks are made; for example, the PC cannot recruit troops if he speaks a different language from the fief (unless he has high stature), and recruitment can only occur in each fief once per season (as indicated by the fief's *hasRecruited* attribute).

As mentioned in *Section 6.2.2*, although the player may specify a number of troops that he wishes to recruit, the numbers of specific troop types that are raised will depend on national recruitment ratios, held in Globals_Server *recruitRatios*, in conjunction with the fief population, and influenced by a random element. Once the recruitment has proceeded and the funds have been deducted from the PC's home treasury, the troops are added to the PC's army (if he was not leading an army, a new one will have been created).

Army leaders are appointed by calling the Army *AssignNewLeader* method (either from SelectionForm to allow selection of an NPC, or directly from the army management screen if the PC is appointing himself leader). This basically updates the leader's *armyID* attribute and the army's *leader* attribute, whilst also performing other necessary actions if required (for example, removing the new leader from the PC's entourage, and removing the new leader from command any other armies).

It was decided that, rather than create a new movement mechanic specifically for an army, the existing character movement methods would be amended and utilised. Essentially, the army is treated in the same manner as an entourage; if an army is detected during the execution of *moveCharacter*, its location will be updated and its days synchronised with those of the leader.

A unique 'housekeeping' mechanic associated with the army object, required for sake of authenticity, is attrition, a process of gradual disintegration that occurs if any army is not properly supplied in the field. This effect is modelled in *Overlord* by applying a formula that takes into account the army's size, in relation to the population of the fief they are in. The chance of attrition occurring is modified by the season and by the leader's attributes (desertions are lessened for a more effective leader), and casualties are also increased during the winter and spring seasons.

Attrition casualties are applied by using the Army *ApplyTroopLosses* method which accepts a double (between 0 - 0.99) and applies losses by iterating through the army's troop types; this method is used to apply all troop losses, irrespective of the circumstance. The code can be seen below (where *lossModifier* is the double passed in):

```
// keep track of total troops lost
uint troopsLost = 0;

for (int i = 0; i < this.troops.Length; i++ )
{
    // calculate no. lost
    uint thisTypeLost = Convert.ToUInt32(this.troops[i] * lossModifier);
    // apply losses
    this.troops[i] -= thisTypeLost;
    // update total
    troopsLost += thisTypeLost;
}

return troopsLost;
```

Attrition is incurred for every new fief that is entered or for every seven days spent in one place (attrition is more likely to occur during periods of exertion). One way to negate the effects of attrition is to maintain the army, paying a set amount per man to provide them with provisions; this can only be done once per season (the army's *isMaintained* attribute being reset during the season update).

Transfers can be made between a player's armies, or between the armies of different players, by using the army management interface to create a detachment of troops and add it to the fief's *troopTransfers* attribute. The detachment can then be picked up (via a screen in SelectionForm), by either the designated recipient PC or the donating PC, and added to the collecting army's own troops. Care was taken to prevent misuse of this mechanic by ensuring that attrition does apply to troop detachments, applied during the season update or when the detachment is picked up.

Due to the multiplayer nature of *Overlord* and its time model, armies can become involved in combat without the player being aware. To allow for this, it was decided to implement a simple 'standing orders' mechanic that allowed an army's behaviour to be specified under certain conditions. This is done in the army management interface through the adjustment of two army attributes, *aggression* and *combatOdds*:

- *aggression* determines the default stance an army takes when faced with combat (for example, an army attempts to stand and fight if the value is 1).
- *combatOdds* is used to modify the default behaviour in some circumstances (for example, the army may only stand and fight if the combat odds are below the specified level).

6.8 Combat

In order to progress toward victory in *Overlord*, a player will usually have to acquire additional fiefs to those he started the game with; combat is the most reliable way to achieve this. Combat comprises of various mechanics, most of which are interconnected to some extent:

- Battle, in which one army attacks another 'in the field'.
- Pillage, in which an army will attempt to extract money from a fief, and to damage its infrastructure; pillage can involve a battle between the pillaging army and the fief garrison and militia.
- Siege, in which an army will attempt to gain control of the fief. This may take several 'rounds' and may involve both battle (if the defending forces sally from the keep) and pillage (if the siege is successful).
- Quell rebellion, in which an army attempts to restore calm in a fief where the populace has risen in rebellion. If successful, this will result in the army owner assuming ownership of the fief, irrespective of who was the original owner.

6.8.1 Battle

The battle mechanic is a fairly complex one, and uses Form1 *GiveBattle* and related methods. Separate methods are employed to:

- Perform various conditional checks prior to attack: Army *ChecksBeforeAttack*).
- Obtain battle values for both armies: Form1 *CalculateBattleValue*.
- See if the defending army has been successfully brought to battle: Form1 *BringToBattle*. This uses data held in Globals_Server *battleProbabilities*.
- Decide whether the attacker is victorious: Form1 *DecideBattleVictory*.

- Calculate casualties (troops) and injuries (characters): Form1 *CalculateBattleCasualties* and Character *CalculateCombatInjury*. The chance of character injury is based on the army casualty level, modified by the character's *combat* attribute.
- Handle retreats: Form1 *CheckForRetreat* and Form1 *ProcessRetreat*

The formula for deciding an army's battle value involves the following steps:

- The calculation of its base combat value: the total combat value for all troops and any characters involved (i.e. the leader and his entourage). Combat values for different troop types of different nationalities are held in Globals_Server *combatValues*.
- The calculation of the army leader's leadership value, which takes into account the leader's *combat*, *management* and *stature* attributes, and the influence of any 'battle' traits.
- The comparison of the two army leadership values to derive a modifier which is then applied to one of the armies.

The calculation of casualties proved to be quite problematic. Initially, this formula applied losses proportionally to each side, taking into account the battle odds and which side had achieved victory; however, testing demonstrated that this often resulted in high losses for a large army, even if it had been victorious against a smaller opponent. It was therefore decided to base casualties on the relative battle values of the two armies (taking into account the size and quality of an army and its leader), using an adapted version of Lanchester's Laws – see Adams (2004). In the case of the largest army losing, the casualties would derive from an approximation of Lanchester's Linear Law (the two armies losing approximately the same numbers); in the case of the largest army winning, the casualties would derive from a modification of Lanchester's Square Law (the largest army suffering the least casualties).

Retreat paths are calculated in a semi-random fashion; if the option is available, the retreating army will always choose to retreat into a fief owned by the army owner. If not, a hex is chosen at random. The starting hex is noted, to ensure that the army does not 'double back' on itself.

Interaction between the battle mechanic and various others, notably siege, meant that particular care had to be taken to ensure that the outcome of the battle did not result in the removal of objects being used by other methods; for example, a battle could result in the loss of a character or even in the ending of a siege. Also, the JournalEntry object created in *GiveBattle* has to account for many different circumstances (battle, siege, or pillage) and outcomes (who won, if any characters were injured or died, if there was a retreat, etc.).

Battle can be affected by the defending army's standing orders. An *aggression* value of 0 will cause the defending army to attempt retreat; so will a value of 1, depending on how the odds for the coming battle compare with the *combatOdds* value.

The outcome of a battle will positively and negatively affect the stature of, respectively, the victorious army's owner and the defeated army's owner.

6.8.2 Pillage

Pillage is a secondary mechanic in *Overlord*; i.e. it is not essential to victory but can help the player to achieve it; it is performed by Form1 *PillageFief* and *ProcessPillage*.

At the start of a pillage, a defending army is created for the fief (using data from Globals_Server *recruitRatios*) consisting of the garrison, a 'professional' force, the size of which is specified by garrison expenditure, and fief militia, which contains a high proportion of untrained 'rabble' and whose size is derived from the fief population. This force will then try to bring the pillaging army to battle, and this may result in the cancelation of the pillage.

The pillage, if it proceeds, will take a variable number of days (7-15) and the outcome is calculated on the basis of the number of pillaging troops per 1000 population; it will adversely affect population, treasury, loyalty, fields, and industry, and will net the pillaging army's owner a variable amount of money, based on the fief GDP.

To realistically reflect the attitudes of the time, however, the pillaging PC will never gain any stature from a pillage; he will actually lose stature if he has pillaged a fief with the same BaseLanguage as himself, and even more if it has the same Language (i.e. the same dialect).

Pillage, like battle, can be affected by standing orders: if the fief that an army intends to pillage contains an enemy army, and if that army has an *aggression* value of 2, the pillage cannot proceed until it has departed the fief (either voluntarily or as the result of battle).

6.8.3 Siege

The decision was taken, given the complexity and possible duration of a siege, to create a Siege class to enable the proper synchronisation of all objects and processes involved, including armies, characters, fief, casualties, and start season and year. Additionally, a new PlayerCharacter attribute, *mySieges*, was introduced to store the sieges in which he is involved (as besieger or defender). A new user interface screen was added which lists all of the player's sieges and allows him to both see information about the siege and, if he is the besieger, to carry out actions (for example, storm the keep).

The main methods associated with conducting a siege are Form1 *SiegeStart*, *SiegeStormRound*, *SiegeReductionRound*, *SiegeNegotiationRound*, and *SiegeEnd*. Also, the Siege *SyncSiegeDays* method was introduced to ensure that the days of all relevant participating objects were synchronised with the days of the besieging army's leader (seen as the controlling entity).

Like pillage, a siege can be prevented by the presence of an enemy army with the appropriate standing orders. If that army is located inside the keep, however, it will be included as a component of the siege (in the Siege *defenderAdditional* attribute). This army may then sally to attack the besieging army if it has the appropriate *combatOdds* value.

Once the siege has commenced, the besieging player can choose to conduct a number of different types of siege 'round', each lasting 10 days:

- The reduction round (Form1 *SiegeReductionRound*) is the default activity for a siege; i.e. the besieging army will attempt to reduce the keep level and inflict defender casualties with a view to storming the keep at a later point. At the beginning of the round, if an additional defending army is present, it may attempt to sally and attack; if successful, the siege will be raised.

- The negotiation round consists of a reduction round, followed by a call to Form1 S*iegeNegotiationRound*. The chance of success is calculated as if for a battle, with each keep level representing 1000 troops for the defender. The resulting percentage value is divided by 2 to derive the chance of a negotiated success. If successful, the siege is ended and ownership changes hands in a civilised manner (using Fief *ChangeOwnership*); the besieging PC will receive an increase in status for the success but will not suffer a penalty for failure.

- The storm round consists of a reduction round, followed by a call to Form1 *SiegeStormRound*. The chance of success is calculated as if for a negotiation, except that the full percentage value is used. A storm round will always result in additional damage to defences and casualties to both defender and attacker, with attacker casualties being even higher in the case of failure.

  As with negotiation, success in a storm round will end the siege and result in a change of fief ownership but there are also additional consequences: firstly, the fief is pillaged and, secondly, captives are taken and ransomed. Only the defending PC and his family, and PCs of a different nationality to that of the besieging PC are taken captive; the ransom consists of the family allowance for NPCs and a proportion of the GDP for PCs (i.e. the total GDP of all of their fiefs). Finally, the besieging PC will have his stature positively or negatively affected depending on the result, the amount being based on the fief's population.

Given the possible duration of a siege, attrition does play a part, especially for the besieging army. The point at which attrition starts to affect the defending forces is dependent on the *management* value of the fief bailiff (who leads the garrison).

The logical consequences of a siege made it necessary to refactor several other mechanics. For example:

- Many army management functions are unavailable to a defending army in a siege.
- Many fief management functions are affected, including the calculation of fief income and the ability to view some financial information.
- A marriage may need to be postponed if one of the couple is inside the keep of a besieged fief.

6.8.4 Quelling a rebellion

Quelling a fief rebellion is a relatively straightforward procedure as it assumes a general state of chaos, rather than the presence of an organised enemy army, and can be a 'cheap' way to gain possession of a fief; it uses Fief *QuellRebellion* and associated methods.

The chance of success is simply a product of the proportion of troops in the quelling army as compared to the fief population, and is modified by the army leader's leadership value. If the army owner is the fief's ancestral owner and the current fief owner is not, this will also have a positive effect on the chance.

Success results in a pillage of the fief and, if he does not already own it, fief ownership being transferred to the army owner. Failure simply results the army having to perform a single hex retreat.

6.9 Second project extension

At this point in the implementation, it became clear that, with some core functionality not yet implemented, an additional extension would be required in order to complete the project to a satisfactory standard.

It was considered unnecessary to amend the previously agreed aims and objectives but, as this project would form the basis for those that followed, it was felt that the reliability of the game engine was important, and that this could best be achieved by carrying out some basic refactoring and testing.

6.10 Royal functions and overlord functions

Kings in *Overlord* can perform certain functions to reward (bribe) followers and cement their position; these gifts involve potential increases in stature and, in most cases, financial gain. Accordingly, a 'Royal gifts' container was created in the user interface, to allow access to these functions, and to display financial information. To facilitate viewing permissions, the *kingOne* and *kingTwo* attributes were added to Globals_Game; heralds (*heraldOne* and *heraldTwo*) also need to have access to royal financial information in their role as royal advisers.

6.10.1 Granting province titles and fief titles

As with all PCs, the king can grant the title of a fief that he owns to any NPC (via the fief management interface). Unlike normal PCs, however, the king can also grant fief titles to other PCs (his loyal subjects) and this is performed through the royal gifts interface, which uses SelectionForm to list potential candidates, before calling Character *TransferTitle* to make the appropriate changes (updating the fief's *titleholder* attribute and the character's *myTitles*).

More importantly, the king can also grant titles of any provinces that he owns, thereby giving the new title holder the opportunity to set and collect taxes from the constituent fiefs. This is done using the same procedure described above.

The king can also, subsequently, revoke a title (province or fief), again by calling *TransferTitle*, and passing himself as the new holder.

6.10.2 Granting fief ownership

The king can also grant ownership of a fief to another PC; as with a successful siege, this is done by using Fief *ChangeOwnership*. Unlike granting a title, however, once fief ownership has changed, it cannot be revoked.

6.10.3 Granting positions

Another feature of *Overlord*, introduced to maintain historical accuracy, was the ability of the king to appoint characters to honorary positions, which would result in an increase of stature. In order to achieve this, the Position object was created, extending Rank, and containing attributes *titleHolder* and *nationality*, which allow the position (unlike a Rank) to be associated with a particular nationality and individual character.

The granting of a position is performed using the Position *BestowPosition* method, which performs the necessary changes to the character's *statureModifer*; like fief or province titles, positions can also be revoked at any time, cancelling the stature enhancement.

6.10.4 Overlord functions

Ironically, overlord functions in *Overlord* are quite limited. A PC can:

- View those provinces for which he is the title holder or the owner.
- Examine the constituent fiefs and see how much tax income they produce.
- Adjust the province tax rate, thereby modifying the income generated.

6.11 Victory conditions and calculation

As mentioned in *Section 5.1.3 Game goals*, an important aspect of any game in motivating players and maintaining their interest is the establishment of suitable victory conditions; in the case of Overlord, these conditions should be both readily understood and reflect historically accurate goals. Three victory conditions were defined during the design phase: total, historically-based, and individual points. After some consideration, it was decided to add an additional victory condition, in which the victors would be whichever players were the kings for either side when the game ended.

Total victory (i.e. one team/nationality owning all fiefs in the game world) was relatively straightforward to implement, involving iteration through the fief master list, whilst keeping a running total of owner nationality.

The historically-based victory, although discussed, was not fully implemented in this project, mainly because the era and geographical scope used in the prototype system did not model a specific historical conflict.

In order to keep a record of individual scores, the VictoryData class was created, containing the player ID and other details associated with the calculation of victory. Scores take into account the following factors:

- The percentage of population falling under the player's control,
- The percentage of total fiefs owned by the player.
- The percentage of total money possessed by the player.
- The player's stature.

In each case, the score is based not only on the final figures but also on the degree of change during the course of the game. It is necessary, therefore, to store both the initial and current values, and to ensure that they are updated during the season update. Accordingly, VictoryData contains methods to calculate the individual components of the score (for example, *CalcStatureScore*) as well as *UpdateData*, which calls various methods in PlayerCharacter to obtain the current data (for example, *GetMoneyPercentage*).

In order to facilitate the achievement of the 'kings victory' and, to a lesser extent, the points victory, it was decided to implement a mechanic that allowed players to challenge for ownership of both provinces and kingdoms, the latter being effectively a bid for the crown. In each case, the criterion for success is the same:

- To achieve ownership of a province, a player needs to own more than 50% of the constituent fiefs for 4 consecutive seasons.
- To achieve ownership of a kingdom, a player needs to own more than 50% of the constituent provinces for 4 consecutive seasons.

The OwnershipChallenge class was created to keep track of current challenges, storing details of the challenger, place ID, place type, and an integer, to be incremented during each season update. Globals_Game *ProcessOwnershipChallenges* updates challenges by iterating through the appropriate master list, and calculating the proportion of constituent parts owned by the challenger; each challenge either has its counter incremented or is removed. If successful (i.e. the counter has reached the value of 4), the *TranferOwnership* method of the appropriate class is used to make the necessary changes, including the update of Globals_game *kingOne* and *kingTwo*, if appropriate.

6.12 System administrator functions

One of the requirements for the game model (see *Section 1.3.1.1*) was provision for a system administrator (sysadmin) role with a full complement of administrative functions. These functions should include the editing of game objects, the population of global variables, the viewing of game logs and event journals, the banning of players from a game, etc. Accordingly, a *sysAdmin* attribute was added to Globals_Game and a container created in the user interface to provide access to the functions.

Due to lack of development time, however, sysadmin functions were only partially realised, being limited to the ability to edit a selection of game objects (PC, NPC, Fief, Province, Kingdom, Army and Trait); attribute validation had not been implemented before development ceased.

6.13 Code refactoring and testing

The final stage of implementation was dedicated to code review and refactoring, and concurrent testing across both Windows and Linux platforms.

Primary code review and refactoring tasks included:

- Ensuring the provision of comprehensive validation in constructors.
- Ensuring the existence in all methods of appropriate checks for null values or empty strings.
- Moving, where possible, of functionality from event-triggered methods (for example, responding to the click of an interface button) into game class methods.
- Moving methods from one class to another, where appropriate.

Testing was separated into two distinct phases:

- Constructor testing, focussing on validation, using the CSV import methods.
- A period of gameplay testing in which all main game mechanics were systematically utilised.

The initial intention was to carry out the main gameplay testing in Linux, in order to make use of the Riak DBMS to populate the game world with a full collection of game objects. However, it became apparent, that the debugging features offered by MonoDevelop were inadequate for the task. As a result, additional methods were created to allow the initiation of a new game through the direct import of CSV data, and the primary testing was switched to the Windows platform.

As was expected in such a complex application, the gameplay testing revealed a selection of errors, unreliable formulas, inconsistent programming practices, and functions that had not been fully implemented. A certain amount of restructuring was therefore required.

Unfortunately, there was simply not enough time to finish moving all methods to their correct classes. To facilitate the future completion of this task, Form1 was sub-divided into partial classes, each of which contained methods associated with a particular area of functionality.

7. REFLECTION

Reflecting upon a large and complex project will necessarily involve contemplation of a broad range of topics, such as the manner in which the project was managed, the suitability of the tools and technologies used, and personal reflections regarding various aspects of my own performance.

7.1 Compliance with requirements, aims and objectives

Given the academic context of the project, the most sensible place to start would be to assess the degree to which the project's aims and objectives have been achieved. This is can be best accomplished through taking a 'bottom up' approach by considering to what extent the requirements outlined in *Section 1.3 Requirements and specification* have been met.

Functional requirements	**Compliance**
Game model	
Contain the following basic objects: PC, NPC, Fief, Army	✓
Provide the following roles: Faction leaders (i.e. kings), Faction heralds, System administrator (sysadmin)	✓
Provide channels of communication	Partial
Be historically accurate within the limitations of game and the level of abstraction	✓
Game engine	
Provide a minimal user interface, through a test client	✓
Allow players to communicate with each other via chat and/or bulletin boards	external
Allow players to register (and delete) accounts	X
Allow players to see games in progress	X
(*Optional*) Design the architecture in such a way as to facilitate the future modification of game content	Partial
System architecture	
Provide a means for storing both persistent game world data, data associated with individual games, and data metrics.	✓
Extra-functional requirements	
Game engine	
Be designed to run on Microsoft Windows (version 7 onwards) and any Linux-based operating system.	✓
(*Optional*) Allow for the future development of a graphical user interface (GUI)	✓
System architecture	
Allow scalability up to thousands of concurrent players	Partial
In the interests of modularity, provide clear interfaces to all game and system components and a clear definition of protocols used	Partial
(*Optional*) provide sufficient redundancy to ensure availability in the event of the failure of key system architecture components	Partial

Table 7: Compliance with project requirements

As can be seen in *Table 7*, the project succeeded in fulfilling many of its core requirements, albeit partially in some cases. However, *Table 8* shows that in some cases the requirements were actually over-achieved through the introduction of features that were initially intended for implementation in future projects.

Feature	**Explanation**
Test client	A more thorough test client was developed to facilitate a supplementary period of testing and code review.
Code quality	Increased code quality, and thus robustness, due to the above.
Game data	The use of more detailed 'game quality' data to allow better improved testing in more realistic game environment.
Ancestral ownership	Mechanic not scheduled for implementation in this game version (see *Section 7.1.3* for details).
Individual troop types	Expanded mechanic not scheduled for implementation in this game version (see *Section 7.1.3*).
Language refinement	Enhanced mechanic not included in planning (see *Section 7.1.3*).
Expenditure adjustment	New mechanic not included in planning (see *Section 7.1.3*).
Troop estimates	Enhanced mechanic not included in planning (see *Section 7.1.3*).

Table 8: Over-achievement of project requirements

7.1.1 Functional requirements compliance

A game model was developed that included all of the stipulated class types (PC, NPC, Fief, Army) and provided the means to assume the key roles (king, herald, system administrator).

Game model design was thorough, systematically addressing both prosaic issues, such as class structure, victory conditions and the flow of game resources, and creative issues, such as the interaction of game mechanics, and character personality. Care was taken to provide players with challenging but attainable goals, whilst also allowing them total freedom to pursue their own actions and agendas, should they wish.

Within the limitations of the game and the level of abstraction, a strong effort has been made to model historical accuracy; this applies to:

- The mechanics available; for example, siege and pillage.
- The rules within which the mechanics operate; for example, the rule that prevents a character granting the title of a fief for which he is the ancestral owner.
- The object data imported into the game, which represent real-life characters and attempt to accurately depict the physical world as it was during the medieval period.
- The formulas employed in key methods, which attempt to model realistic consequences; for example, the use of Lanchester's Laws to model battlefield casualties.

The requirement to provide channels of communication was only partially implemented when development ceased. A system had been developed that allowed the communication of in-game events (such as birth, siege outcomes, etc.) and the necessary modifications had been made to allow the creation of messages for all key game events. However, although trialled during the testing phase, this functionality had not been extended to the collection of game metrics.

A test client was developed which provides the player with an interface with which to interact with the game world and its constituent objects, within the limits defined by the game rules. Crucially, due to a period of testing and limited refactoring, the *JominEngine* is robust.

An effort was made to facilitate the future modification of game content by storing key data, such as troop strengths and recruitment ratios in globally accessible variables, although this was not fully implemented due to insufficient time. The documentation of game formulas, showing the methods in which they appear (see *Appendix B: Formulas*), should expedite the completion of this process.

As specified in the initial plan, no full client-server was implemented, rather just an embedded test-client. Therefore, the game engine functional requirements that were dependant on a full implementation were not completed; namely, the facility to communicate directly with other players using chat or bulletin boards, the provision of user accounts and related functions, and the display of game progress information. However, the decision had already been made to investigate external options in some of these cases (for example, chat and account login).

Through the implementation of an interface to the *Riak* DBMS, a means was provided for storing both game world and other game-related data, although only the storage of game world data was actually implemented. Limited testing of *Riak* would appear to support its suitability for the task, although this cannot be confirmed until more rigorous testing under full game conditions has been performed.

7.1.2 Extra-functional requirements compliance

As specified, the *JominEngine* supports both *Microsoft Windows* and, through the use of *Mono*, *Linux*-based operating systems.

The decision to use the *Microsoft .Net* framework, *Mono* and particularly the choice of *C#* as the programming language, should facilitate the porting of the *JominiEngine* into *Unity 3D*, a cross-platform game creation system, which will allow the future implementation of a full graphical user interface.

As client-server was not implemented, only partial progress towards supporting scalability; specifically, the decision to use the *Riak* DBMS was made both because of its purported ability to sustain performance during periods of high usage, and its distributed nature, which could potentially allow game data to be stored locally, thereby reducing latency for a widely distributed player base. Similarly, although no explicit fault-tolerance mechanisms were introduced in the game engine code, the provision of system redundancy was partially addressed through the choice to use *Riak* which, being distributed, has a high degree of built-in redundancy.

Where possible, an effort has been made to provide clear interfaces to game and system components and to ensure the provision of adequate documentation; for example, there is a distinct interface to the DBMS, and its use of JSON for serialisation ensures full supporting documentation. However, some work still needs to be done regarding the final location of game methods; to facilitate their future removal to other game classes, Form1 partial classes have been created, containing methods associated with specific areas of functionality.

7.1.3 Requirements over-achievement

Table 8 details those areas of the implementation which have resulted in over-achievement.

In order to facilitate a supplementary period of testing and code review, not originally included in the planning, a more thorough test client was developed. Additionally, the use of historically accurate 'game quality' data allowed testing to be carried out in a true game environment. These measures, in turn, led to enhanced code quality and, therefore, a more robust product.

In some cases, an expanded feature was added ahead of schedule, or a completely new feature introduced; examples include:

- Ancestral ownership: a PC's ancestral ownership of a fief can have an important effect upon his interaction with other game objects (for example, his status can be enhanced). This mechanic was not scheduled for implementation until a later project.
- Individual troop types: it was initially decided to have a single troop type (foot) and to introduce other types with varying combat values subsequently. However, I implemented the full complement of troop types in this project.
- Refined language definition: the interaction between character and fief languages can have an effect upon the character's ability to perform certain tasks (e.g. recruitment) or upon his status (when pillaging a fief). I have refined this concept to allow a distinction to be made between the base language (say, English) and the exact language, which also includes dialect. This should provide for increased realism.
- Expenditure adjustment: this is a useful mechanic that allows a player to automatically adjust fief expenditure (in proportion with original expenditure levels) to match funds available in the fief treasury. It can be accessed manually but is also performed automatically during the season update. This is a new mechanic, not mentioned at all in planning.
- Troop estimates: this models the uncertainty involved with estimating the number of troops in an enemy army. It bases the accuracy of the estimate upon the observer's military skills (leadership value, including traits), determining a 'scope of error', and then using a random number to arrive at an estimate.

7.1.4 Compliance with aims and objectives

Aims	**Compliance**
Develop a core game model, specifying the basic game classes (PC, NPC, Fief, and Army), interactions, rules, mechanisms, and victory metrics	✓
Develop a core game engine, enabling players to perform the functionality outlined in the game model	✓
Develop an underlying system architecture, enabling the efficient operation of the game engine at varying usage levels	Designed (not implemented)
Primary objectives	
Creation of an interactive core game engine	✓
Modularity: to enable future expansion by ensuring that the game engine be designed with clear, well documented interfaces and protocols to all game components	Partial
Scalability: ensuring that the game will support up to several thousand concurrent players, at an acceptable level of performance	Partial

Secondary (optional) objectives	
Content authoring: exposure of game data to facilitate the modification of game content	Partial
Availability: providing a sufficient level of redundancy to allow for the failure of key system architecture components	Partial

Table 9: Compliance with project aims and objectives

In summary, as can be seen in *Table 9*, through the achievement of most core requirements, the majority of the project's aims and objectives have also been fully or partially realised; those that were not fully accomplished, were focussed around the unfortunate failure to implement the underlying client-server architecture.

Significantly, the project was able to add to the scope of knowledge in the area of MMORPG design by proposing and examining the novel use of a NoSQL (and therefore non-standard) DBMS and, at least in principal, demonstrating its suitability for the task; the main research into this aspect, however, will need to be undertaken in future projects.

The most important achievement was the implementation of a robust core game engine that allowed players to participate in the historically authentic game world defined in the game model, and to interact with a realistic collection of game objects in ways defined by rational but unrestrictive rules.

7.2 Project management

During the course of the project, I feel it is in the area of project management that I learned the most pertinent lessons.

Throughout the project, I attempted to follow the suggested guidelines in the *British Computer Society's Code of good practice*. Prior to commencing the project, meetings were held at which aims, objectives and deliverables were discussed, and a timeline agreed; I decided to take an agile approach to the project, which would ensure both flexibility and timely communication and feedback.

I was generally satisfied with the way the project was managed:

- The risk assessment accurately identified potential problems and, in each case, provided a course of action that could be taken to rectify them.
- The agile approach seemed to work well, with good communication in both directions and rapid feedback on progress.
- On the whole record keeping was good, notably with regard to the use of Git commit messages to record the frequent changes to the code. The only area that could have been improved was the recording of meeting minutes, which were often written rather than typed.
- Despite the need for extensions, the project progressed in methodical and logical way, sequentially focussing on functional areas identified in the design phase.

- The testing and code refactoring stage proved very fruitful in identifying errors and potential weaknesses and enabled the production of a more robust product.

One issue raised in *Section 4.3.4 Social issues*, is the need to be aware of potentially offensive game features. There are a small number of features in *Overlord* that may fall under this category; however, in each case care was taken to consider whether their inclusion was warranted within the context of the project. For example, the assertion that English troops were superior to their opponents can be justified by historical accuracy. Another example, the reduced role of women, reflects to a certain extent historical circumstances, but was also a deliberate design decision, in order to reduce complexity in the initial version; it may be rectified in a future extension.

I was also aware, however, of those aspects of the project management where I made mistakes:

- My lack of experience in undertaking projects of this size meant that the initial aims and objectives were overly ambitious; realistically, attempting to develop both core game engine and client-server architecture was a step too far.

- Another consequence of my inexperience was the failure to comprehend that a seemingly simple application could actually become very complex over the course of a project; towards the end of implementation, the introduction of even a small feature often resulted in numerous changes to existing code.

- I also did not appreciate the extent to which my employment would affect the time available to work on the project, especially when taken in conjunction with my own deliberate nature which tended to lengthen the amount of time spent on the various stages.

- In some cases, my enthusiasm for the project resulted in the development of functionality that had not been agreed in the initial plan. For example, although it had been agreed to include attributes to show ancestral ownership of fiefs, the associated mechanics were not due to be implemented in this project; *however, I decided to do so*. This, and similar decisions, would later have an impact on the time available for testing and refactoring.

However, I realise that the realisation of one's mistakes is a valuable part of the process, and I would certainly be wiser regarding these matters in any future projects.

7.3 Programming practices

On the whole, I was pleased with the solid (if unsophisticated) programming technique that I applied during the course of implementation. I feel I also exhibited a reasonable degree of flexibility when faced with the challenges of a complex design; in particular, the frequent code revision necessary upon the addition of a new mechanic.

I was careful throughout implementation to make use of XML summaries for classes, attributes and methods (giving details of parameters and returned objects), and I included copious comments to explain the flow and intent of the code.

Although limited time was available for testing and code review, it was carried out in a systematic manner, the results were properly documented (see *Appendix D: Testing*), and it proved beneficial to quality of the application.

However, it is necessary to also recognise some areas in which a combination of inexperience and the pressure of time led to mistakes being made.

On occasion, my inexperience resulted in indecision that slowed implementation and sometimes resulted in a lack of consistency; for example, my uncertainty over whether to use embedded objects or object IDs in attributes. There are advantages and disadvantages for both practices: the use of full objects gives more immediate access to object data and reduces the need to iterate through object master lists. However, it requires that more work be carried out to ensure the serialisation of those attributes and, by extension, this has an impact on the time required to carry out database read/write operations. Ultimately, it will probably depend on the way in which the database is utilised during the game.

Also, despite attempting where possible to break methods into component, easily manageable parts, there are nevertheless a small number of very large methods (for example, Character *ProcessDeath* and Form1 *GiveBattle*) that will need to be sub-divided.

I optimistically assumed I would have several weeks in which I could review and refactor code. Thinking that I could rely on this stage at the end of the implementation phase, I sometimes pushed ahead with code, knowing that it did not conform to good practice but believing that I would have sufficient time to put it right. As is quite a common occurrence during software projects, however, time ran out before these changes could be made. For example, various methods were created in the Form1 class, simply to provide easy access to all game objects; many of these should probably be located in game object classes.

7.4 Tools and technologies used

The overall performance and suitability of tools and technologies selected for use on the project was high.

Both *Visual Studio 2010* and *MonoDevelop* offered a good range of features, including straightforward refactoring (for example, method or attribute renaming), good layout that made for easy navigation of the application code, and plugins for version control systems, including *Git*. Unfortunately, *MonoDevelop*'s debugging facilities left a lot to be desired, making it very hard to track errors as they occurred, and often resulting in the need to reboot. As a result, Windows became the de facto development platform, even when coding for the Linux-based *Riak* DBMS.

Mono proved to be a very reliable distribution of the *.NET* framework, supporting the full functionality of *.NET* 4.5 without any noticeable performance issues. It was especially useful for the development of the interface to the *Riak* DBMS.

The chosen programming language, *C#*, was fully featured and offered an extensive collection of useful libraries, as would be expected for a language developed specifically for Windows by Microsoft. Both of the external libraries selected for the project, *QuickGraph* (for creating the graph-based game map) and *CorrugatedIron* (for creating the *Riak* interface) were easy to use and contained some very useful methods. Additionally, JSON, used by *CorrugatedIron* for serialisation, might prove to be a suitable protocol for client-server communication in later projects.

Riak was easy to install and straightforward to use, offering both port-based and HTTP-based connectivity. It appeared to perform with acceptable speed and without any significant impact upon the game engine, even when loading thousands of game objects upon game start-up. However, it was only used in a rudimentary fashion in this project, and will require further investigation in a client-server environment under more strenuous conditions before a decision can be made regarding its suitability.

The *Git* version control system, along with its web-based repository service *GitHub*, proved an invaluable tool during the implementation stage of the project. It provided a number of useful functions, including the ability to:

- Share the *JominiEngine* code.
- Create parallel code branches to test different features.
- Examine code from previous versions.
- Create incremental comments detailing code changes, thereby enabling the implementation history to be traced.
- Continue working with a local copy of the repository during the occasional period when an Internet connection could not be obtained.
- Act as a cloud-based backup.

UML (Unified Modelling Language) diagrams were frequently used in this report to enable the graphical representation of various aspects of the *JominiEngine* design, including basic structure (class diagram), inter-object communication (sequence diagram), game world 'work flows' (activity diagram) and 'interaction space' between the player and game mechanics (use case diagram).

I have found UML to be very useful, both during the design and implementation phases, for visualising what is essentially an entirely abstract entity (a software program) from a number of different perspectives, and allowing the components of that program to be comprehended irrespective of the programming language being used to implement it.

During the design of the game model, however, I occasionally encountered a concept that was difficult to illustrate using UML; for example, the interaction between the various processes involved with the fief management mechanic (a compound mechanic). I wished to show how the individual elements affected each other, sometime directly, sometimes inversely; ultimately, I found that a non-UML entity relationship diagram best suited my purposes.

On a related note, I can recommend the open source utility *UMLet* for the creation of the basic types of UML diagrams with the minimum of fuss.

7.5 Required changes to the existing code

As previously mentioned, although the core game engine is largely completed, some tidying up is required before future projects can be undertaken:

- Some methods are too large, making the logic of the code hard to follow; these need to be sub-divided into more manageable methods, each addressing a distinct function. Primary examples include Form1 *GiveBattle* and *SiegeStormRound*, and Character *ProcessDeath*.

- Form1 has become a 'God class' containing methods that should be located in other classes. These methods have been placed into Form1 partial classes (for example, Form1_battle) to facilitate this process.

- Object validation needs to be added to the object editing methods accessible via the system administrator interface (Form1 *SaveTraitEdit*, *SaveArmyEdit*, *SaveCharacterEdit*, and *SavePlaceEdit*).

There are also some other, optional, tasks that should be considered:

- The methods used for error reporting are not entirely consistent, with constructors making use of exceptions but methods in the database interface (for example, Form1 *DatabaseRead_NPC*) using popup MessageBoxes to report errors; these should probably be changed to use exceptions.

- The process of using global variables to store data employed in key formulas was only partially completed (for example, Globals_Server *combatValues*); this should to be completed to facilitate the future modification of game content. *Appendix B* contains information on the formulas used and the methods in which they appear.

- Currently, even if a character dies in the game, their object remains in the master list (to allow access to the data, if required). However, over the course of a long game, this will have an inevitable impact on performance, and so an alternative method needs to be implemented (for example, keeping this data in the backend database).

7.6 Suggestions for future development

The ultimate goal of this and subsequent projects is to produce a game engine that will allow the creation of fully featured historically-based MMORPGs set in a medieval context. This project has made a solid start through the development of the core game functionality but considerable work remains to be done before this goal is achieved.

7.6.1 Suggestions for short-term development

7.6.1.1 Game system architecture

An obvious starting point for future development would be the implementation of those elements that were originally planned for this project but which were not completed. Specifically, the implement of client-server architecture and the associated multiplayer functions, namely player communication (for example, via chat or bulletin board), player accounts (including login), and lobby functions (for examples, view games in progress, join an existing game). Care should be taken to examine the use of third-party solutions in order to reduce system complexity and development time.

Once this infrastructure is in place, investigations can be carried out into the crucial areas of scalability and redundancy. The performance of the *Riak* DBMS should be thoroughly tested to ascertain its suitability, and comparisons should be made between its single node and distributed modes. *Riak* 2 is now available (current version 2.0.4) and its functionality should be investigated.

The issue of basic load-balancing should be addressed, including the possibility of mirroring (sharding) or sub-division of the game world over separate servers. If the latter option is examined, strategies will need to be developed for the transition of game objects between game world regions and for coping with game objects and events that occur in the border zone of two regions (and are therefore potentially visible to players in both regions).

7.6.1.2 Game design

There are also extensions that could be made to the game design, especially regarding the expansion or completion of existing mechanics. In my opinion, these changes would be best carried out subsequent to the introduction of the client-server architecture mentioned above.

- The completion of the system administrator functions to include, for example, the ability to ban a player, and to re-assign roles for a game in progress. Some of these functions will be necessary if the multiplayer architecture is implemented.
- The expansion of the data collection/communication mechanic to include metrics allowing the analysis of various aspects of game usage. This data could also be examined in conjunction with latency measurements, etc.
- The addition of new character attributes to 'flesh out' the personality and allow a more fine-tuned interaction with the game world. These might include the re-introduction of attributes simplified for the initial version of the game, such as leadership, or the introduction of completely new attributes, such as intelligence.
- In conjunction with the above, the expansion of the traits collection from its current limited subset.
- The expansion of the geographical scope to include France, Scotland, and Ireland, converting 'megafiefs' where necessary.
- The introduction of a sea travel mechanic. This will be necessary if the suggested changes to geography above are implemented. Also, sea travel can be examined within the context of dividing the game world between separate servers.
- The implementation of the Hundred Years War scenario.

Additional suggestions for expanding existing mechanics that would be relatively straightforward to implement include:

- Expanded roles for women, allowing them to be PCs, and to inherit.
- Expanded combat, to model the effects both of terrain and of one troop type on another.
- Expanded pillage to allow the lesser option of raiding.
- Expanded barring to allow the barring of characters speaking particular languages.
- Expanded roles for NPCs, including spying and recruitment.
- Expanded troop transfers to allow the joining of two armies, or the separation of one.
- Expanded inheritance to allow more complex family trees.
- Expanded use of the distinction between Language and BaseLanguage into the areas of recruitment and bailiff effectiveness (currently, the distinction is only applied during pillage).

7.6.2 Suggestions for medium-term development

7.6.2.1 Game system architecture

Section 1.1, Aims and objectives, and *Section 1.3, Requirements and specifications*, alluded to the possible future introduction of several features, including the modification of game content, and a functional graphical user interface (GUI).

Game content modification relies on the exposure of key game data in order to allow game content, and sometimes game behaviour, to be customised by the player. A start has already been made in this area (for example, the ability to import game objects from CSV files) but further work needs to be carried out both on the *JominiEngine* data structure and on provision of a suitable interface. Third-party options should be thoroughly investigated, such as the use of *Lua* scripting language, well-regarded by the game developer community.

Probably the most sensible route to the introduction of a GUI for the *JominiEngine* would the use of a third-party game development system, negating the requirement for the in-house production of high quality graphics and accompanying changes to infrastructure. *Unity 3D*, one such system that is popular in the game developer community, should be investigated to ascertain its suitability. Crucially, certain decisions regarding the technology used to develop the *JominiEngine* were taken with this system in mind; specifically, *Unity* is built on Mono and uses C# as one of its main development languages.

The prior implementation of client-server architecture would also provide further development opportunities. Investigation can be carried out into the separation of functions such as the administration of network connection to players, or the segregation of database storage in order to provide faster access to dynamic data.

Strategies for reducing network traffic could also be examined, such as the introduction of the area of interest (AOI) concept, requiring that players only be updated on changes to game state that are relevant to them. However, this would only apply if the *JominiEngine* implements a form of 'fog of war'.

7.6.2.2 Game design

Short-term changes to the game design (see *Section 7.6.1.2*) suggested expansions to already existing mechanics. The next stage would be to introduce new mechanics into the *JominiEngine*. These might include:

- The introduction of game elements based on the role of religion, a very important part of medieval life. Examples include the ability for PCs to join a crusade; the possibility of being excommunicated for wrongdoings (in the eyes of the church); the introduction of divorce and the role played in it by the church. A new role of pope might be implemented, giving access to some of these functions.
- 'Dirty tricks' such as assassination, kidnapping, and seduction as methods by which a player can destabilise his opponents' positions and raise some extra money. The combination of these mechanics and religion would add an extra dimension to the game, as well as increased authenticity.
- The release of a series of well-researched scenarios depicting specific historical conflicts such as the Wars of the Roses and the invasion of England by the Normans. This would also compliment, and possibly be used as a proof of concept for, the introduction of a content authoring system.
- The ability to train characters based on the attributes of the trainer.
- The introduction of tournament combat as a 'mini game' in which PCs could participate to attain extra stature.
- The introduction of a game mode that disables the royal right of inheritance in cases where a PC has no heir. Instead, the deceased PC's fiefs could fall into rebellion and be claimed through the quelling mechanic.

7.6.3 Suggestions for long-term development

Obviously, long-term development will very much depend on the steps that have been taken previously. However, my suggestions for further enhancements would include:

- The introduction of enhanced NPC AI, facilitating more authentic behaviour in the game world and allowing the autonomous performance of responsibilities such as fief management. This not only adds to the player's sense of immersion but also opens up the project to a completely new area of research and development.
- The creation of 'plug-in' modules to reduce the level of abstraction and significantly expand certain game mechanics; for example, modules that allow battles to be fought on a tactical level, or fiefs to be improved and administered in detail. This would provide the opportunity to potentially extend the game's target player base.
- The expansion of game world geography to include all of Europe. This could be done in a modular fashion, perhaps in conjunction with the release of historical scenarios.
- The introduction of enhanced game graphics and advanced GUI.

8. CONCLUSIONS

8.1 Game model and engine

A game model was developed that included all of the stipulated class types (PC, NPC, Fief, and Army), provided the means to assume key roles (king, herald, system administrator), and specified component mechanics, rules and victory conditions. This fulfils the project's first aim (see *Section 1.1.1*).

The design process for the game model was both thorough and systematic, and resulted in a model that addressed all of the issues specified in the aims and objectives. UML-style diagrams were used extensively for both the game model and the software system design, helping to provide a sound, well-documented basis for the longer term effort.

Care was taken to not only consider functional aspects of the game – object types, mechanics, resources, and rules – but also to ensure that the model reflected, within the limitations of the game, a high degree of authenticity and historical accuracy. One characteristic feature of *Overlord* is the concrete embedding of the game model into a historically accurate context. This is achieved by the choice of rules and game mechanics reflecting the "Age of Magna Carta", but also through the use of a historically accurate database of NPCs and a complete model of England as the geographic context. Within this context, victory conditions were deliberately balanced to offer goals that were both challenging and realistic, whilst also permitting freedom to pursue individual motivations.

Using this game model, a core game engine (the *JominiEngine*) was developed and implemented that allowed players to use the specified mechanics to manipulate objects within the game world in the manner defined by the rules. This fulfils the project's second aim and the first primary objective (see *Section 1.1.2*).

All aspects of the game model were implemented, including some mechanics that had been earmarked for future expansions; however, lack of time resulted in only a partial implementation of secondary game concepts, such as the in-game information messaging system and the system administrator functions.

A prototype client was implemented to allow the testing of the *JominiEngine*; this facilitated a period of code review and testing at the end of the implementation phase, which helped to identify errors and potential weaknesses and enabled the production of a more robust product. Some preliminary work will be required, however, primarily code restructuring, to prepare the *JominiEngine* for future projects.

It is worth noting the, with this prototype client and the above historically accurate context, *Overlord* represents a ***deployable*** instance of the *JominiEngine*, which is an appreciable over-achievement of the originally specified aims of the project

8.2 System architecture, scalability and availability

Although considered during the planning phase, the limited time available meant that the implementation of the client-server aspects of the game engine could not be carried out. Consequently, various system functions that were dependant on the presence of the client-server architecture could not be implemented; specifically, the facility to communicate directly with other players using chat or bulletin boards, the provision of user accounts and related functions, and various ‘lobby’ functions. For these aspects, however, external software, such as chat clients, can be easily used; therefore the lack of these features inside of the game engine doesn't pose a serious limitation to its functionality.

Some progress was made, however, towards the investigation of both system scalability, a primary objective (see *Section 1.1.2*), and system availability and redundancy, a secondary objective; this was achieved through the installation of the *Riak* DBMS, as a means for storing both game state and other game-related data (although only the storage of game state data was implemented in this project). The *Literature review* (see *section 2.2.5*) identifies concrete advantages of *Riak* (and NoSQL databases in general) as a DBMS for MMORPGs.

Riak was chosen for both its purported ability to sustain performance during periods of high usage, and its distributed nature, which could potentially allow game data to be stored locally, thereby reducing latency for a widely distributed player base. Additionally, *Riak*’s distributed architecture also provides a built-in redundancy in the event that one of its nodes should fail. Limited testing of *Riak* would appear to support its suitability for the task; it performed with acceptable speed and without any significant impact upon the game engine, even when reading and writing thousands of game objects. However, it was only employed in a rudimentary fashion in this project, and its distributed architecture remains untested. More rigorous investigation in a client-server environment will be required before a decision can be made regarding its suitability.

Importantly, the use of non-relational (NoSQL) DBMS’s in MMOs is relatively novel and is an area that could benefit from additional research. It is hoped, therefore, that this project was able to add, in a modest fashion, to the scope of knowledge in this particular area of MMORPG design.

8.3 Modularity

With a view to the future implementation of additional functionality, an effort was made to address the primary objective of modularity (see *Section 1.1.2*) through the provision of thorough documentation and clear interfaces to game and system components; for example, there is a distinct interface to the DBMS and to CSV import methods. Where applicable for the purpose, existing protocols have been used; for example, the JSON format is employed by the *CorrugatedIron* library for serialisation, and may prove suitable for client-server communication in future versions of the *JominiEngine*.

XML summary sections have been added to all methods, giving details of parameters and returned objects, and the methods contain copious comments to explain their content. Additionally, the appendices to this report contain details of key formulas used in methods (see *Appendix B: Formulas*), and a list of containers used in the test user interface (see *Appendix D: Testing*).

8.4 Facilitation of future extensions: Content authoring and graphical user interface

Although not fully implemented due to insufficient time, an effort was made to facilitate the future modification of game content (‘modding’), and to increase the flexibility of the game engine, by exposing key data in globally accessible variables, such as troop strengths, battle probabilities and recruitment ratios. The documentation of key formulas used in methods (see *Appendix B: Formulas*), should expedite the completion of this process.

One of the extra-functional requirements for the project (see *Section 1.3.2.1*) was provision for the future development of a graphical user interface (GUI). This has, to a certain extent, been fulfilled through the decisions to use the *Microsoft .Net* framework, *Mono* and *C#*. These choices should facilitate the porting of the *JominiEngine* into *Unity 3D*, a popular game creation system, built on *Mono* and using *C#* as one of its main development languages.

8.5 Suggestions for future development

8.5.1 System architecture

A good starting point for future development would be the introduction of those elements that were not implemented for this project; specifically, client-server architecture and the associated multiplayer functions. The use of external software and third-party solutions should be examined in order to reduce system complexity and development time; for example, *Facebook Login* for player account management, and chat clients such as *mIRC* and *Mystic BBS* for player communication.

Technically the most profitable direction for further work would be to carry out research into scalability and redundancy, with particular reference to the performance of the *Riak* DBMS and its distributed architecture. *Riak* 2 is now available (current version 2.0.4) and its functionality should be investigated.

The issue of basic load-balancing should be addressed, including the possibility of mirroring (sharding) or sub-division of the game world over separate servers. If the latter option is examined, strategies will need to be developed for the transition of game objects between game world regions (servers) and for coping with game objects and events that occur in the border zone of two regions.

In the medium term, further research could be carried out into areas such as the reduction of network traffic; this might include, for example, the introduction of the area of interest (AOI) concept, requiring that players only be updated on changes to game state that are relevant to them.

As referred to in *Section 1.1.3* of *Aims and objectives*, the *JominiEngine* could be extended to allow the modification of game content, although further work needs to be carried out both on the *JominiEngine* data structure before this can be attempted. Third-party options should be thoroughly investigated, such as the use of the well-regarded *Lua* scripting language.

Also alluded to in this project, the implementation of a GUI for the *JominiEngine* would be a necessary step towards the development of fully featured MMORPG. The use of a third-party game development system, removing the need for the in-house production of high quality graphics and accompanying changes to infrastructure, should be investigated; *Unity 3D*, would seem to be an obvious candidate.

8.5.2 Game model and engine

A starting point for future development of the game model and engine would be the expansion or completion of existing mechanics. Specifically, extension of the system administrator functions would be necessary in order to efficiently manage massive multiplayer instances of the engine, and the expansion of the game analytics mechanic to include metrics allowing the analysis of various aspects of game usage, could also prove very useful with regard to measuring system performance.

Other relatively simple short-term enhancements would include the development of a more realistic character model through the expansion of attributes and traits, the expansion of the game world geography (probably requiring the introduction of a sea transport mechanic), and the full implementation of concrete historical episodes in the general time-frame, such as the post-Magna-Carta civil war.

An attempt should be made to increase authenticity of the game model by expanding the roles for women (such as allowing women to inherit - 'Salic Law'), modelling combat in a more realistic fashion (allowing for the effects both of terrain and of one troop type on another), and expanding inheritance through the modelling of complex family trees.

Medium-term developments should concentrate on introducing new mechanics into the *JominiEngine*. Two such areas, that would complement each other well, would be:

- The introduction of the role of religion, a very important part of medieval life (see Steele, 2012). Religious mechanics might include the ability to join a crusade, the modelling of investiture and its potential political conflicts, the possibility of being excommunicated, and the introduction of divorce. In this context, the new role of a Pope should be modelled and implemented.
- 'Dirty tricks' such as assassination, kidnapping, and seduction as methods by which a player can destabilise his opponents' positions and raise some extra money.

A way to demonstrate the versatility of the *JominiEngine* might be to release of a series of well-researched scenarios depicting specific historical conflicts. This could be carried out in conjunction with, and possibly be used as a proof of concept for, the introduction of a content authoring system.

Long-term development might explore the area of enhanced NPC artificial intelligence, which could be employed to model authentic behaviour in the game world, and allow the autonomous performance of responsibilities such as fief management. This not only adds to the player's sense of immersion but also opens up the project to a completely new area of research and development.

Another way to extend the game model, and to potentially extend the game's player base, would be to create 'plug-in' modules that allow a switch from the current strategic management of game components, to more detailed tactical management; for example, modules that allow battles to be fought on a tactical level, or fiefs to be improved and administered in detail.

9. REFERENCES

Aarseth, E., Smedstad, S.M. and Sunnana, L. (2003) 'A multi-dimensional typology of games' in: *Level Up: Digital Games Research Conference 2003*, Utrecht: The Netherlands, 48-53.

Adams, E. (2004) The Designer's Notebook: Kicking Butt by the Numbers: Lanchester's Laws [online], available: http://www.gamasutra.com/view/feature/2123/the_designers_notebook_kicking_.php [accessed 9 February 2015].

Alecu, V.M. (2012) *Developing a client-server architecture and minimizing data transfer for a massively multiplayer online game*, MSc dissertation Utrecht University, unpublished.

Alqwbani, A., Zuping, Z. and Aqlan, F. (2014) 'Big Data Management for MMO Games and Integrated Website Implementation', *Global Journal of Computer Science and Technology (B): Cloud and Distributed*, 14(2).

Alverez, J. et al. (2006) 'Morphological study of the video games', in *Proceedings of the International Conference on Games Research and Development (CGIE '06)*, pp. 36–43, Perth, Australia, December 2006.

Anderson, E.F. et al. (2010) 'Developing serious games for cultural heritage: a state-of-the-art review', *Virtual Reality*, 14(4), December 2010, 255-275.

Anderson, E.F. (2011) 'A classification of scripting systems for entertainment and serious computer games', in *Proceedings: 2011 Third International Conference on Games and Virtual Worlds for Serious Applications: VS-Games 2011,* Los Alamitos: IEEE Computer Society, 47-54.

Andrade, G., & Corruble, V. (2005) 'Challenge-sensitive action selection an application to game balancing', in *Proceedings: The 2005 IEEE/WIC/ACM International Conference on Intelligent Agent Technology*, Los Alamitos: IEEE Computer Society, 194–200.

Assiotis, M. and Tzanov, V. (2005) 'A Distributed Architecture for MMORPG', in *NetGames '06: Proceedings of 5th ACM SIGCOMM Workshop on Network and System Support for Games*, New York: ACM, Article No. 4., 1-7.

Avedon, E. M. (1971) 'The Structural Elements of Games', In Avedon, E. M. & Brian Sutton-Smith eds., *The Study of Games*, New York: John Wiley and Sons, 419-426.

Bartle, R. (1996) 'Hearts, clubs, diamonds, spades: Players who suit MUDs', *Journal of Virtual Environments* [online], 1(1), available: http://mud.co.uk/richard/hcds.htm [accessed 5 April 2014].

Bharambe, A. (2006) 'Colyseus: A distributed architecture for online multiplayer games', in *Proceedings: Symposium on Networked Systems Design and Implementation (NSDI)*, San Jose: USENIX, 3-6.

Bharambe, A. et al. (2008) 'Donnybrook: enabling large-scale, high-speed, peer-to-peer games', in *SIGCOMM '08: Proceedings of the ACM SIGCOMM 2008 conference on Data communication*, New York: ACM, 389-400.

Binstock, A. (2013) The Quiet Revolution in Programming [online], available: http://www.drdobbs.com/architecture-and-design/the-quiet-revolution-in-programming/240152206 [accessed 5 April 2014].

Broadberry, S., Campbell, B. and van Leeuwen, B. (2011) 'English Medieval Population: Reconciling time series and cross sectional evidence.' *University of Warwick unpublished manuscript*. [online], available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.360.9324&rep=rep1&type=pdf [accessed 9 February 2015].

Buyukkaya, E. and Abdallah, M. (2008) 'Data management in Voronoi-based P2P gaming', in *Proceedings: CCNC 2008: 5th IEEE Consumer Communications and Networking Conference*, Los Alamitos: IEEE Computer Society, 1050-1053.

Buyukkaya, E. et al. (2009) 'VoroGame: A hybrid P2P architecture for massively multiplayer games', in *Proceedings: CCNC 2009, 6th IEEE Consumer Communications and Networking Conference*, Alamitos: IEEE Computer Society, 1 - 5.

Caltagirone, S. et al. (2002) 'Architecture for a Massively Multiplayer Online Role Playing Game engine', *Journal of Computing Sciences in Colleges*, 18(2), 105–116.

Crawford, C. (1984) *The art of computer game design* [online], available: http://www.scribd.com/doc/140200/Chris-Crawford-The-Art-of-Computer-Game-Design [accessed 9 February 2015].

Crawford, C. (2002) *The art of interactive design: A euphonious and illuminating guide to building successful software*, San Francisco: No Starch Press.

Crawford, C. (2013) *Chris Crawford on interactive storytelling*, 2nd ed., Berkeley: New Riders.

Cronin, E. et al. (2004) 'An efficient synchronization mechanism for mirrored game architectures', *Multimedia Tools and Applications*, 23(1), 7-30.

Diao, Z. et al. (2014) 'CloudCraft: Cloud-based data management for MMORPGs', in *Databases and Information Systems VIII*, Amsterdam, IOS Press, 71-84.

Dieckmann, M. (2013) A Journey into MMO Server Architecture [online], available: http://www.mmorpg.com/blogs/FaceOfMankind/052013/25185_A-Journey-Into-MMO-Server-Architecture [accessed 5 April 2014].

Drain, B. (2008) EVE Evolved: EVE Online's server model [online], available: http://massively.joystiq.com/2008/09/28/eve-evolved-eve-onlines-server-model/ [accessed 5 April 2014].

Drain, B. (2011) The industry's obsession with shards [online], available: http://massively.joystiq.com/2011/03/29/the-soapbox-the-industrys-obsession-with-shards/ [accessed 5 April 2014].

Driel, M. et al. (2011) A Survey on MMOG System Architectures [online], available: http://kaidence.org/research/MMOG.pdf [accessed 5 April 2014].

Dunnigan, J.F. (2000) W*argames handbook: How to play and design commercial and professional wargames,* 3rd ed., Lincoln: Writers Club Press.

Emilsson, K. (2014) Infinite space: An argument for single-sharded architecture in MMOs [online], available:
http://www.gamasutra.com/view/feature/132563/infinite_space_an_argument_for_.php?print=1
[accessed 5 April 2014].

Fan, L. et al. (2010) 'Design issues for Peer-to-Peer Massively Multiplayer Online Games' *International Journal of Advanced Media and Communication*, 4(2), 108-125.

Gaitatzes, A. et al. (2004) 'The Ancient Olympic Games: being part of the experience', in *VAST '04: Proceedings of the 5th International Conference on Virtual Reality, Archaeology and Cultural Heritage*, Aire-la-Ville: Eurographics Association, 19–28.

Gillingham, J. (1990) *The Wars of the Roses: Peace and conflict in fifteenth-century England*, London: Weidenfeld and Nicolson.

Hu, S-Y. et al. (2008) 'Voronoi state management for peer-to-peer massively multiplayer online games', in *Proceedings: CCNC 2008: 5th IEEE Consumer Communications and Networking Conference*, Los Alamitos: IEEE Computer Society, 1134-1138.

Iimura, T. et al. (2004) 'Zoned federation of game servers: a peer-to-peer approach to scalable multi-player online games', in *NetGames '04: Proceedings of the 3rd ACM SIGCOMM Workshop on Network and System Support for Games*, New York: ACM, 116-120.

Jacobson, J. and Holden, L. (2005) 'The Virtual Egyptian Temple', in *Proceedings: EdMedia 2005: World Conference on Educational Multimedia, Hypermedia & Telecommunications*, Chesapeake: AACE, 4531-4536.

Jardine, J. and Zappala, D. (2008) 'A hybrid architecture for massively multiplayer online games', in *NetGames '08: Proceedings of the 7th ACM SIGCOMM Workshop on Network and System Support for Games*, New York: ACM, 60-65.

Jinzhonh, W. A. N. G. and Zhigang, Y. U. E. (2010) 'A finding less-load server algorithm based on MMOG and analysis', in *International Conference on Intelligent Computation Technology and Automation (ICICTA), Changsha, Hunan, China*, 1, 96–99.

Kim, H-Y. and Park, H-J. (2013) 'An efficient gaming user oriented load balancing scheme for MMORPGs' *Wireless Personal Communications*, 73(2), 289-297.

Kirschenbaum, M. (2011) War; what is it good for? Learning from wargaming [online], available: http://www.playthepast.org/?p=1819 [accessed 9 February 2015].

Knutsson, B. et al. (2004) 'Peer-to-peer support for massively multiplayer games', in *INFOCOM 2004: Proceedings of the 23rd Annual Joint Conference of the IEEE Computer and Communications Societies: Volume 1*, Los Alamitos: IEEE Computer Society, 96-107.

Lankosk, P. and Björk, S. (2008) 'Character-driven game design: Characters, conflict, and gameplay', paper for *GDTW 2008: the 6th International Game Design and Technology Workshop and*

Conference, Liverpool [online], available: http://www.academia.edu/175264/Character-Driven_Game_Design_Characters_Conflict_and_Gameplay [accessed 9 February 2015]

Llopis, N. (2009) Data-oriented design (or why you might be shooting yourself in the foot with OOP) [online], available: http://gamesfromwithin.com/data-oriented-design [accessed 5 April 2014].

Martin, A. (2007) Entity Systems are the future of MMOG development – part 3 [online], available: http://t-machine.org/index.php/2007/12/22/entity-systems-are-the-future-of-mmog-development-part-3/ [accessed 5 April 2014].

McKnight, C. et al. (2012) 'Multi-server MMO middleware: Unlocked', in *3PGCIC 2012: Proceedings of the 7th International Conference on P2P, Parallel, Grid, Cloud and Internet Computing*, Los Alamitos: IEEE Computer Society, 218-225.

Muhammad, Y. (2011) *Evaluation and Implementation of Distributed NoSQL Database for MMO Gaming Environment*, MSc dissertation Uppsala University, unpublished.

Nicole, D. (1996) *Medieval warfare source book: Volume I: Warfare in western Christendom*, London: BCA.

Nofi, A.A. and Dunnigan, J.F. (1997) Medieval life & The Hundred Years War [online], available: http://www.macs.hw.ac.uk/~hwloidl/hundredyearswar.com/Books/History/1_help_c.htm [accessed 9 February 2015].

Oman, C.W.C. (1885) *The art of war in the Middle Ages: A.D. 378-1515*, Oxford: B.H. Blackwell

Radoff, John (2007) Anatomy of an MMORPG [online], available: http://radoff.com/blog/2008/08/22/anatomy-of-an-mmorpg/ [accessed 5 April 2014].

Ross, S.J. (2013) Medieval demographics made easy [online], available: http://www222.pair.com/sjohn/blueroom/demog.htm [accessed 5 April 2014].

Sicart, M. (2008) 'Defining game mechanics', *Game Studies: The International Journal of Computer Game Research* [online], 8(2), December 2008, available: http://gamestudies.org/0802/articles/sicart [accessed 5 April 2014].

Steele, L.J. (2012) *Fief: A look at medieval society from its lower rungs*, 2nd ed., Austin: Cumberland Games & Diversions.

Steele, L.J. (2010) *Town: A city-dweller's look at thirteenth to fifteenth century Europe*, Austin: Cumberland Games & Diversions.

Sumption, J. (1990) *The Hundred Years War: Volume I: Trial by battle*, London: Faber and Faber.

Sumption, J. (1999) *The Hundred Years War: Volume II: Trial by fire*, London: Faber and Faber.

Sumption, J. (2009) *The Hundred Years War: Volume III: Divided houses*, London: Faber and Faber.

Tuchman, B.W. (1979) *A distant mirror: The calamitous 14th century*, Harmondsworth: Penguin.

Van Geel, I. (2013) MMOData.net: Keeping track of the MMORPG scene [online], available: http://mmodata.blogspot.co.uk/ [accessed 5 April 2014].

Von Clausewitz, C. (1997) *On war*, Ware: Wordsworth Editions Limited.

Von Reiswitz, G.H.R. (1824) *Instructions for the representation of tactical maneuvers under the guise of a wargame*, s.l.: s.n.

West, M. (2007) Evolve your hierarchy: Refactoring game entities with components [online], available: http://cowboyprogramming.com/2007/01/05/evolve-your-heirachy/ [accessed 5 April 2014].

White, W. et al. (2009) 'Better scripts, better games', *Communications of the ACM: Being Human in the Digital Age*, 52(3), March 2009, 42-47.

Yee, N. (2006) 'Motivations of Play in Online Games', *CyberPsychology & Behavior*, December 2006, 9(6), 772-775.

Yonekura, T. et al. (2004) 'Peer-to-peer networked field-type virtual environment by using AtoZ', in *CW 2004: Proceedings: 2004 International Conference on Cyberworlds*, Los Alamitos: IEEE Computer Society, 241-248.

Zeigler, B. (2006) Why SQL sucks for MMORPGs [online], available: http://doublebuffered.com/2006/10/30/why-sql-sucks-for-mmorpgs/ [accessed 5 April 2014].

APPENDIX A: DIAGRAMS

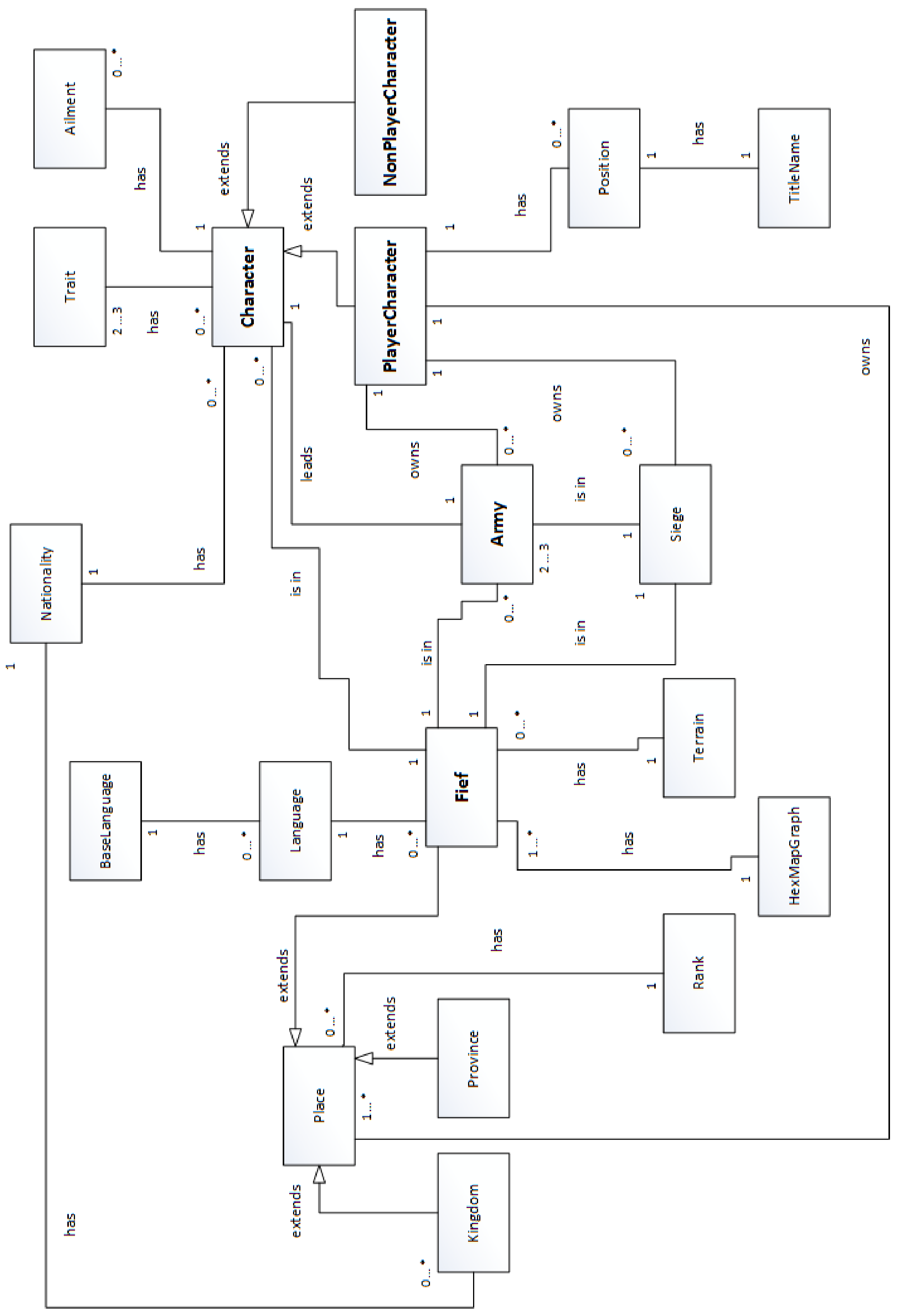
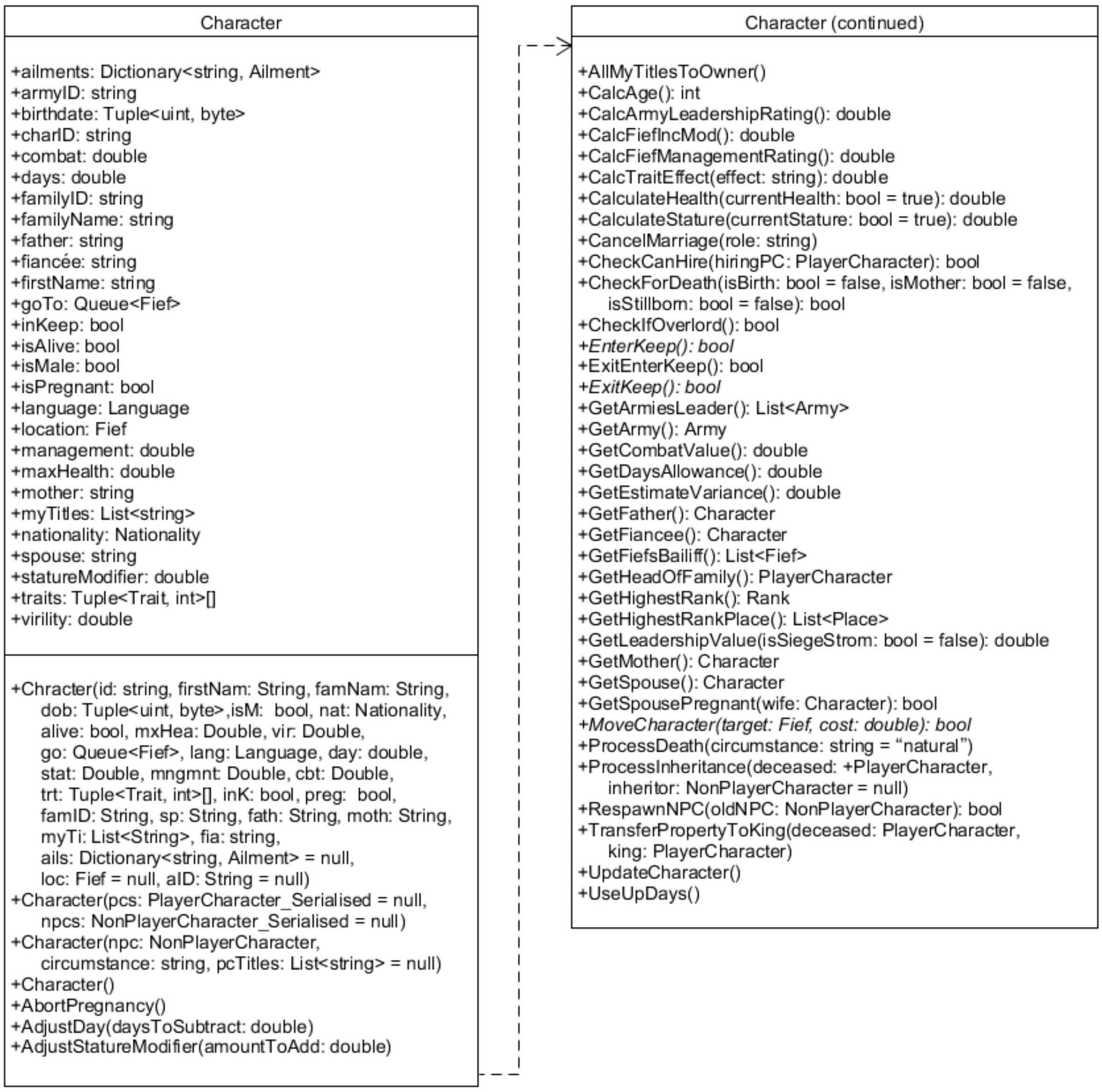


Figure 13: Class entity relationship diagram showing main JominiEngine classes

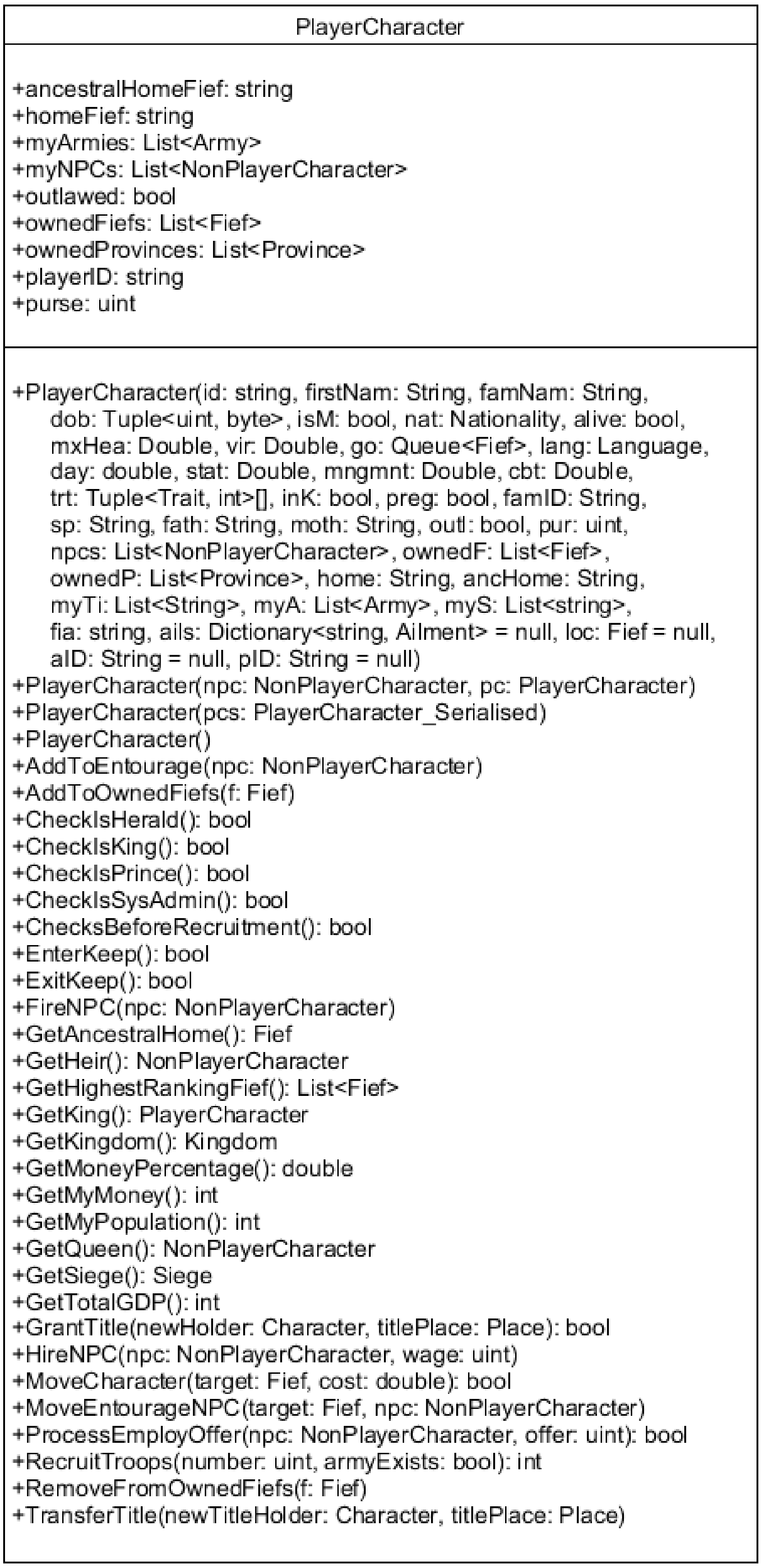


```
Character
+ailments: Dictionary<string, Ailment>
+armyID: string
+birthdate: Tuple<uint, byte>
+charID: string
+combat: double
+days: double
+familyID: string
+familyName: string
+father: string
+fiancée: string
+firstName: string
+goTo: Queue<Fief>
+inKeep: bool
+isAlive: bool
+isMale: bool
+isPregnant: bool
+language: Language
+location: Fief
+management: double
+maxHealth: double
+mother: string
+myTitles: List<string>
+nationality: Nationality
+spouse: string
+statureModifier: double
+traits: Tuple<Trait, int>[]
+virility: double

+Chracter(id: string, firstNam: String, famNam: String,
    dob: Tuple<uint, byte>,isM:  bool, nat: Nationality,
    alive: bool, mxHea: Double, vir: Double,
    go: Queue<Fief>, lang: Language, day: double,
    stat: Double, mngmnt: Double, cbt: Double,
    trt: Tuple<Trait, int>[], inK: bool, preg:  bool,
    famID: String, sp: String, fath: String, moth: String,
    myTi: List<String>, fia: string,
    ails: Dictionary<string, Ailment> = null,
    loc: Fief = null, aID: String = null)
+Character(pcs: PlayerCharacter_Serialised = null,
    npcs: NonPlayerCharacter_Serialised = null)
+Character(npc: NonPlayerCharacter,
    circumstance: string, pcTitles: List<string> = null)
+Character()
+AbortPregnancy()
+AdjustDay(daysToSubtract: double)
+AdjustStatureModifier(amountToAdd: double)
```

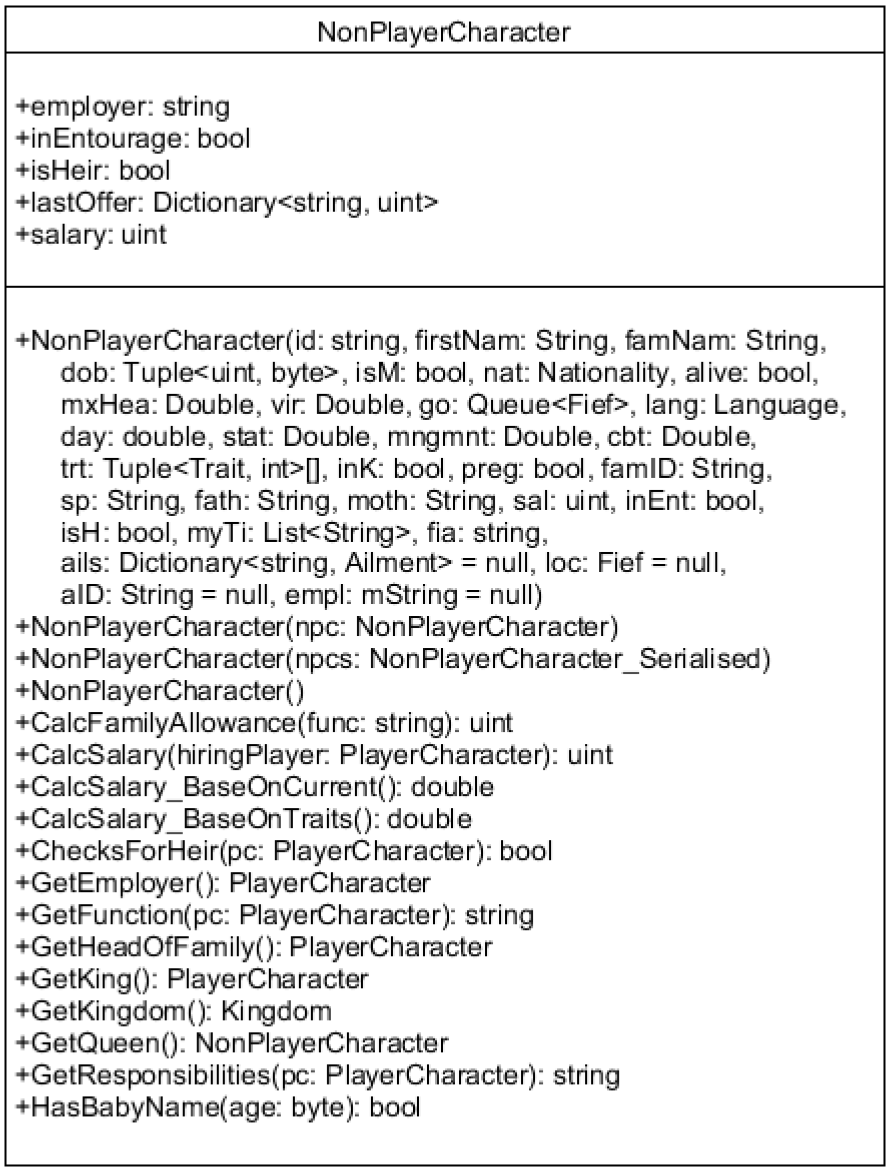
```
Character (continued)
+AllMyTitlesToOwner()
+CalcAge(): int
+CalcArmyLeadershipRating(): double
+CalcFiefIncMod(): double
+CalcFiefManagementRating(): double
+CalcTraitEffect(effect: string): double
+CalculateHealth(currentHealth: bool = true): double
+CalculateStature(currentStature: bool = true): double
+CancelMarriage(role: string)
+CheckCanHire(hiringPC: PlayerCharacter): bool
+CheckForDeath(isBirth: bool = false, isMother: bool = false,
    isStillborn: bool = false): bool
+CheckIfOverlord(): bool
+EnterKeep(): bool
+ExitEnterKeep(): bool
+ExitKeep(): bool
+GetArmiesLeader(): List<Army>
+GetArmy(): Army
+GetCombatValue(): double
+GetDaysAllowance(): double
+GetEstimateVariance(): double
+GetFather(): Character
+GetFiancee(): Character
+GetFiefsBailiff(): List<Fief>
+GetHeadOfFamily(): PlayerCharacter
+GetHighestRank(): Rank
+GetHighestRankPlace(): List<Place>
+GetLeadershipValue(isSiegeStrom: bool = false): double
+GetMother(): Character
+GetSpouse(): Character
+GetSpousePregnant(wife: Character): bool
+MoveCharacter(target: Fief, cost: double): bool
+ProcessDeath(circumstance: string = "natural")
+ProcessInheritance(deceased: +PlayerCharacter,
    inheritor: NonPlayerCharacter = null)
+RespawnNPC(oldNPC: NonPlayerCharacter): bool
+TransferPropertyToKing(deceased: PlayerCharacter,
    king: PlayerCharacter)
+UpdateCharacter()
+UseUpDays()
```

Figure 14: Class diagram showing the Character and Trait classes



```
PlayerCharacter
--------------------------------------------------------------
+ancestralHomeFief: string
+homeFief: string
+myArmies: List<Army>
+myNPCs: List<NonPlayerCharacter>
+outlawed: bool
+ownedFiefs: List<Fief>
+ownedProvinces: List<Province>
+playerID: string
+purse: uint
--------------------------------------------------------------
+PlayerCharacter(id: string, firstNam: String, famNam: String,
    dob: Tuple<uint, byte>, isM: bool, nat: Nationality, alive: bool,
    mxHea: Double, vir: Double, go: Queue<Fief>, lang: Language,
    day: double, stat: Double, mngmnt: Double, cbt: Double,
    trt: Tuple<Trait, int>[], inK: bool, preg: bool, famID: String,
    sp: String, fath: String, moth: String, outl: bool, pur: uint,
    npcs: List<NonPlayerCharacter>, ownedF: List<Fief>,
    ownedP: List<Province>, home: String, ancHome: String,
    myTi: List<String>, myA: List<Army>, myS: List<string>,
    fia: string, ails: Dictionary<string, Ailment> = null, loc: Fief = null,
    aID: String = null, pID: String = null)
+PlayerCharacter(npc: NonPlayerCharacter, pc: PlayerCharacter)
+PlayerCharacter(pcs: PlayerCharacter_Serialised)
+PlayerCharacter()
+AddToEntourage(npc: NonPlayerCharacter)
+AddToOwnedFiefs(f: Fief)
+CheckIsHerald(): bool
+CheckIsKing(): bool
+CheckIsPrince(): bool
+CheckIsSysAdmin(): bool
+ChecksBeforeRecruitment(): bool
+EnterKeep(): bool
+ExitKeep(): bool
+FireNPC(npc: NonPlayerCharacter)
+GetAncestralHome(): Fief
+GetHeir(): NonPlayerCharacter
+GetHighestRankingFief(): List<Fief>
+GetKing(): PlayerCharacter
+GetKingdom(): Kingdom
+GetMoneyPercentage(): double
+GetMyMoney(): int
+GetMyPopulation(): int
+GetQueen(): NonPlayerCharacter
+GetSiege(): Siege
+GetTotalGDP(): int
+GrantTitle(newHolder: Character, titlePlace: Place): bool
+HireNPC(npc: NonPlayerCharacter, wage: uint)
+MoveCharacter(target: Fief, cost: double): bool
+MoveEntourageNPC(target: Fief, npc: NonPlayerCharacter)
+ProcessEmployOffer(npc: NonPlayerCharacter, offer: uint): bool
+RecruitTroops(number: uint, armyExists: bool): int
+RemoveFromOwnedFiefs(f: Fief)
+TransferTitle(newTitleHolder: Character, titlePlace: Place)
```

Figure 15: Class diagram showing the PlayerCharacter class

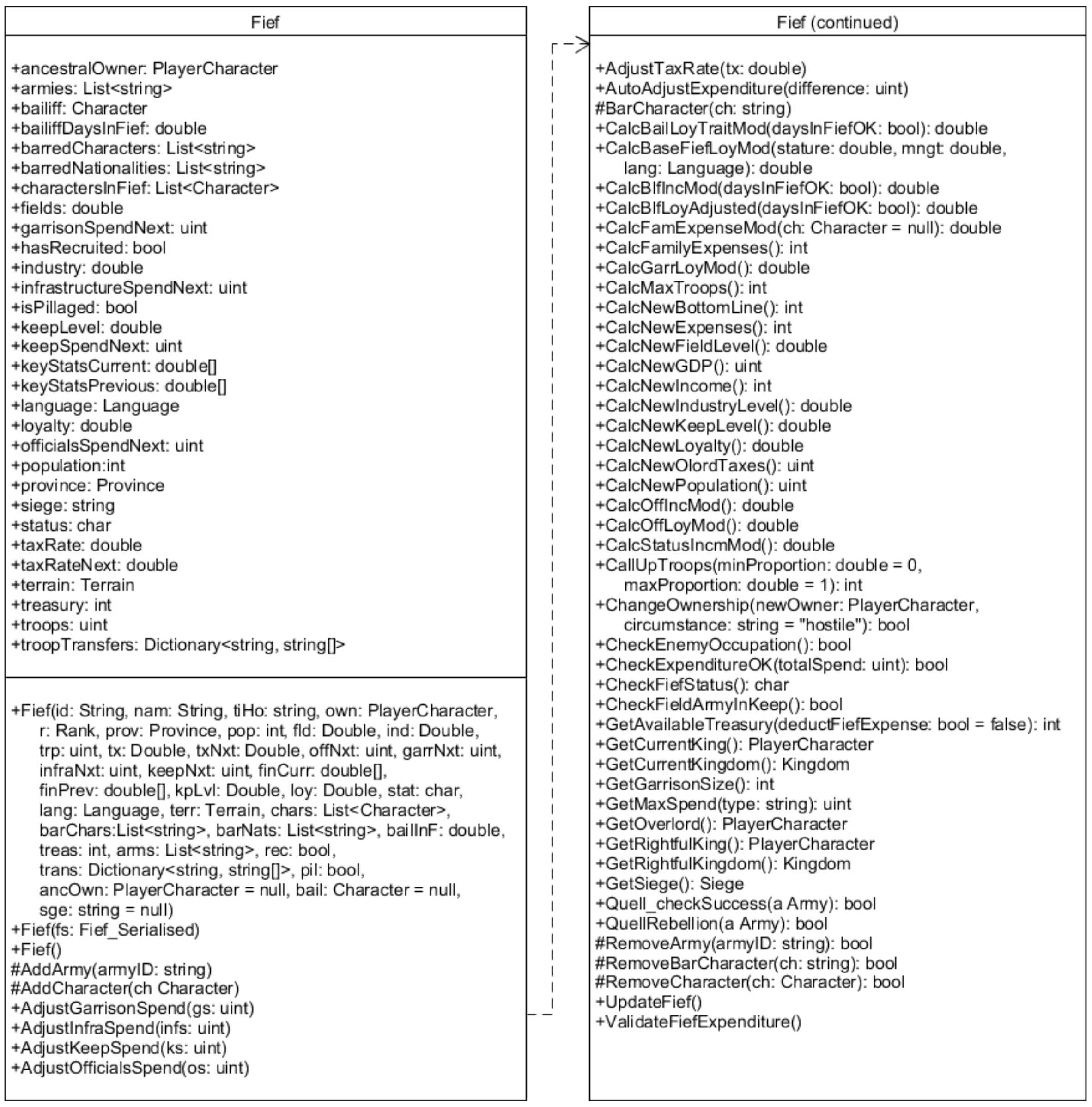


```
NonPlayerCharacter

+employer: string
+inEntourage: bool
+isHeir: bool
+lastOffer: Dictionary<string, uint>
+salary: uint

+NonPlayerCharacter(id: string, firstNam: String, famNam: String,
    dob: Tuple<uint, byte>, isM: bool, nat: Nationality, alive: bool,
    mxHea: Double, vir: Double, go: Queue<Fief>, lang: Language,
    day: double, stat: Double, mngmnt: Double, cbt: Double,
    trt: Tuple<Trait, int>[], inK: bool, preg: bool, famID: String,
    sp: String, fath: String, moth: String, sal: uint, inEnt: bool,
    isH: bool, myTi: List<String>, fia: string,
    ails: Dictionary<string, Ailment> = null, loc: Fief = null,
    aID: String = null, empl: mString = null)
+NonPlayerCharacter(npc: NonPlayerCharacter)
+NonPlayerCharacter(npcs: NonPlayerCharacter_Serialised)
+NonPlayerCharacter()
+CalcFamilyAllowance(func: string): uint
+CalcSalary(hiringPlayer: PlayerCharacter): uint
+CalcSalary_BaseOnCurrent(): double
+CalcSalary_BaseOnTraits(): double
+ChecksForHeir(pc: PlayerCharacter): bool
+GetEmployer(): PlayerCharacter
+GetFunction(pc: PlayerCharacter): string
+GetHeadOfFamily(): PlayerCharacter
+GetKing(): PlayerCharacter
+GetKingdom(): Kingdom
+GetQueen(): NonPlayerCharacter
+GetResponsibilities(pc: PlayerCharacter): string
+HasBabyName(age: byte): bool
```

Figure 16: Class diagram showing the NonPlayerCharacter class



```
Fief

+ancestralOwner: PlayerCharacter
+armies: List<string>
+bailiff: Character
+bailiffDaysInFief: double
+barredCharacters: List<string>
+barredNationalities: List<string>
+charactersInFief: List<Character>
+fields: double
+garrisonSpendNext: uint
+hasRecruited: bool
+industry: double
+infrastructureSpendNext: uint
+isPillaged: bool
+keepLevel: double
+keepSpendNext: uint
+keyStatsCurrent: double[]
+keyStatsPrevious: double[]
+language: Language
+loyalty: double
+officialsSpendNext: uint
+population:int
+province: Province
+siege: string
+status: char
+taxRate: double
+taxRateNext: double
+terrain: Terrain
+treasury: int
+troops: uint
+troopTransfers: Dictionary<string, string[]>

+Fief(id: String, nam: String, tiHo: string, own: PlayerCharacter,
    r: Rank, prov: Province, pop: int, fld: Double, ind: Double,
    trp: uint, tx: Double, txNxt: Double, offNxt: uint, garrNxt: uint,
    infraNxt: uint, keepNxt: uint, finCurr: double[],
    finPrev: double[], kpLvl: Double, loy: Double, stat: char,
    lang: Language, terr: Terrain, chars: List<Character>,
    barChars:List<string>, barNats: List<string>, bailInF: double,
    treas: int, arms: List<string>, rec: bool,
    trans: Dictionary<string, string[]>, pil: bool,
    ancOwn: PlayerCharacter = null, bail: Character = null,
    sge: string = null)
+Fief(fs: Fief_Serialised)
+Fief()
#AddArmy(armyID: string)
#AddCharacter(ch Character)
+AdjustGarrisonSpend(gs: uint)
+AdjustInfraSpend(infs: uint)
+AdjustKeepSpend(ks: uint)
+AdjustOfficialsSpend(os: uint)
```

```
Fief (continued)

+AdjustTaxRate(tx: double)
+AutoAdjustExpenditure(difference: uint)
#BarCharacter(ch: string)
+CalcBailLoyTraitMod(daysInFiefOK: bool): double
+CalcBaseFiefLoyMod(stature: double, mngt: double,
    lang: Language): double
+CalcBlfIncMod(daysInFiefOK: bool): double
+CalcBlfLoyAdjusted(daysInFiefOK: bool): double
+CalcFamExpenseMod(ch: Character = null): double
+CalcFamilyExpenses(): int
+CalcGarrLoyMod(): double
+CalcMaxTroops(): int
+CalcNewBottomLine(): int
+CalcNewExpenses(): int
+CalcNewFieldLevel(): double
+CalcNewGDP(): uint
+CalcNewIncome(): int
+CalcNewIndustryLevel(): double
+CalcNewKeepLevel(): double
+CalcNewLoyalty(): double
+CalcNewOlordTaxes(): uint
+CalcNewPopulation(): uint
+CalcOffIncMod(): double
+CalcOffLoyMod(): double
+CalcStatusIncmMod(): double
+CallUpTroops(minProportion: double = 0,
    maxProportion: double = 1): int
+ChangeOwnership(newOwner: PlayerCharacter,
    circumstance: string = "hostile"): bool
+CheckEnemyOccupation(): bool
+CheckExpenditureOK(totalSpend: uint): bool
+CheckFiefStatus(): char
+CheckFieldArmyInKeep(): bool
+GetAvailableTreasury(deductFiefExpense: bool = false): int
+GetCurrentKing(): PlayerCharacter
+GetCurrentKingdom(): Kingdom
+GetGarrisonSize(): int
+GetMaxSpend(type: string): uint
+GetOverlord(): PlayerCharacter
+GetRightfulKing(): PlayerCharacter
+GetRightfulKingdom(): Kingdom
+GetSiege(): Siege
+Quell_checkSuccess(a Army): bool
+QuellRebellion(a Army): bool
#RemoveArmy(armyID: string): bool
#RemoveBarCharacter(ch: string): bool
#RemoveCharacter(ch: Character): bool
+UpdateFief()
+ValidateFiefExpenditure()
```

Figure 17: Class diagram showing the Fief class

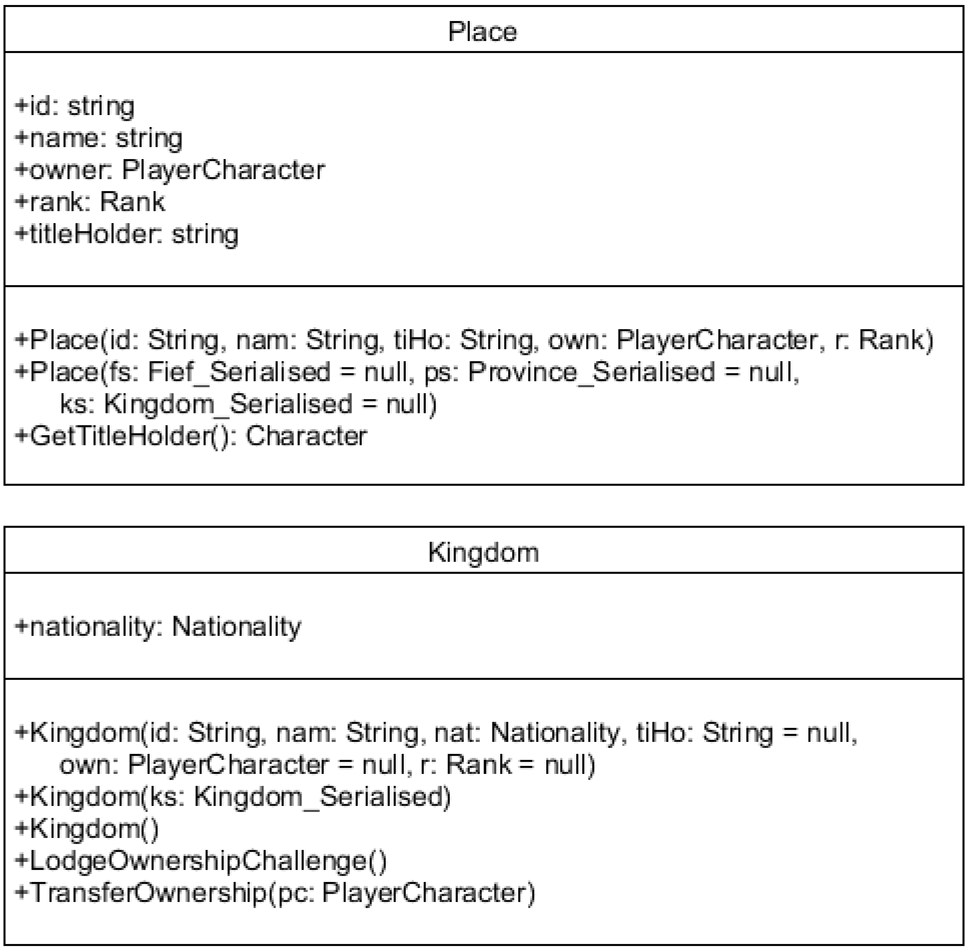

Place
+id: string
+name: string
+owner: PlayerCharacter
+rank: Rank
+titleHolder: string
+Place(id: String, nam: String, tiHo: String, own: PlayerCharacter, r: Rank)
+Place(fs: Fief_Serialised = null, ps: Province_Serialised = null, ks: Kingdom_Serialised = null)
+GetTitleHolder(): Character
Kingdom
+nationality: Nationality
+Kingdom(id: String, nam: String, nat: Nationality, tiHo: String = null, own: PlayerCharacter = null, r: Rank = null)
+Kingdom(ks: Kingdom_Serialised)
+Kingdom()
+LodgeOwnershipChallenge()
+TransferOwnership(pc: PlayerCharacter)

Figure 18: Class diagram showing the Place and Kingdom classes

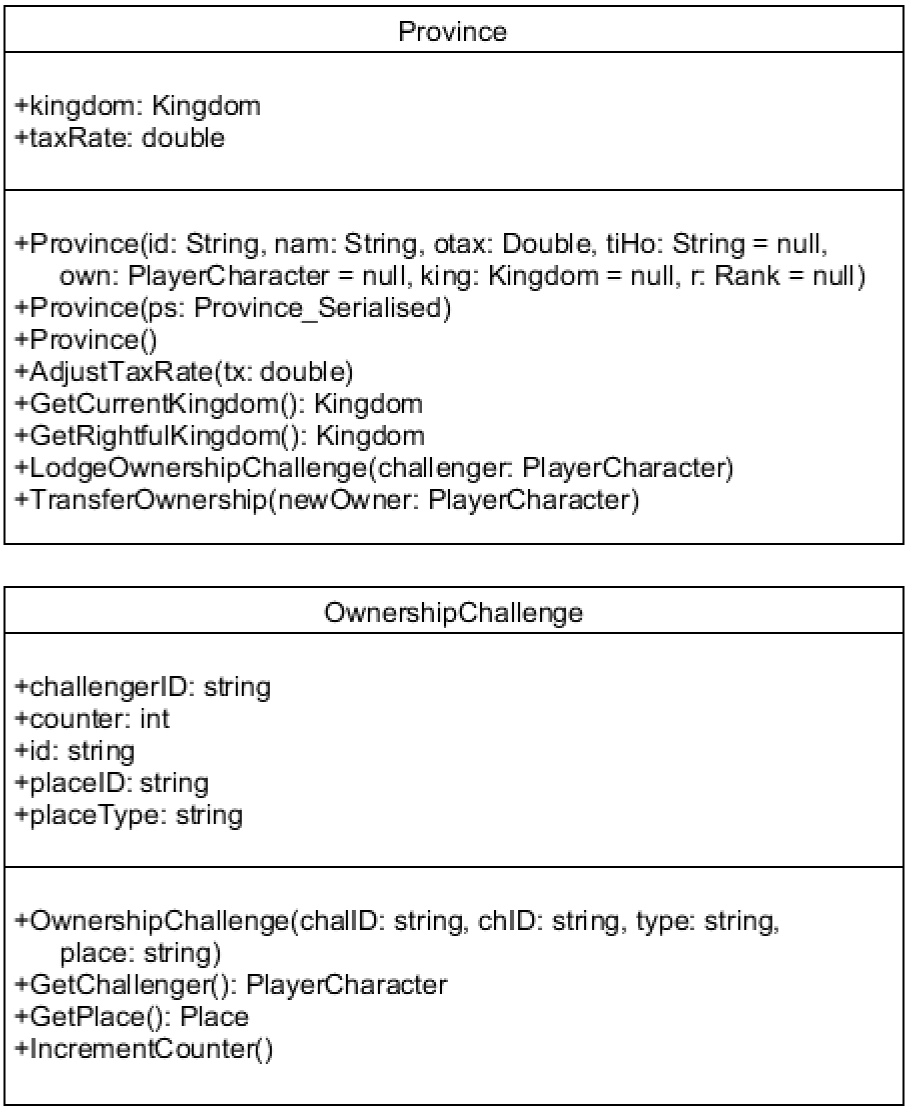
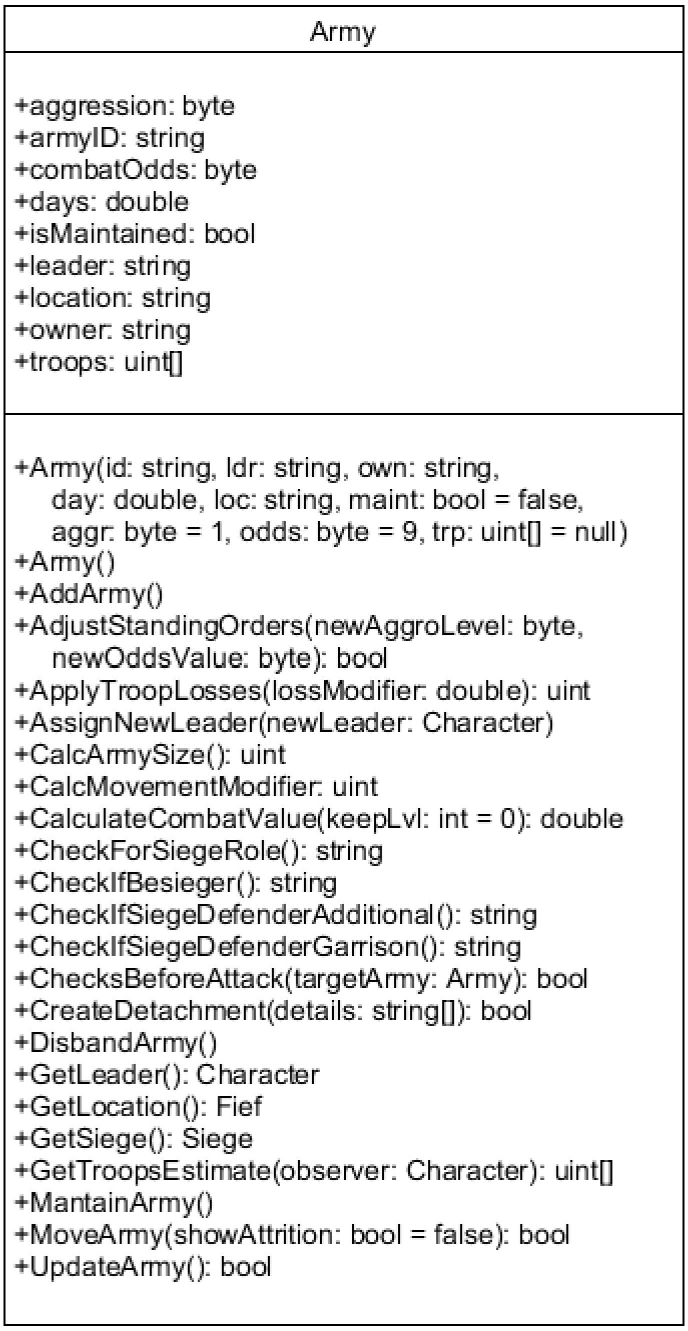

Province
+kingdom: Kingdom
+taxRate: double
+Province(id: String, nam: String, otax: Double, tiHo: String = null, own: PlayerCharacter = null, king: Kingdom = null, r: Rank = null)
+Province(ps: Province_Serialised)
+Province()
+AdjustTaxRate(tx: double)
+GetCurrentKingdom(): Kingdom
+GetRightfulKingdom(): Kingdom
+LodgeOwnershipChallenge(challenger: PlayerCharacter)
+TransferOwnership(newOwner: PlayerCharacter)
OwnershipChallenge
+challengerID: string
+counter: int
+id: string
+placeID: string
+placeType: string
+OwnershipChallenge(chalID: string, chID: string, type: string, place: string)
+GetChallenger(): PlayerCharacter
+GetPlace(): Place
+IncrementCounter()

Figure 19: Class diagram showing the Province and OwnershipChallenge classes



```
Army

+aggression: byte
+armyID: string
+combatOdds: byte
+days: double
+isMaintained: bool
+leader: string
+location: string
+owner: string
+troops: uint[]

+Army(id: string, ldr: string, own: string,
    day: double, loc: string, maint: bool = false,
    aggr: byte = 1, odds: byte = 9, trp: uint[] = null)
+Army()
+AddArmy()
+AdjustStandingOrders(newAggroLevel: byte,
    newOddsValue: byte): bool
+ApplyTroopLosses(lossModifier: double): uint
+AssignNewLeader(newLeader: Character)
+CalcArmySize(): uint
+CalcMovementModifier: uint
+CalculateCombatValue(keepLvl: int = 0): double
+CheckForSiegeRole(): string
+CheckIfBesieger(): string
+CheckIfSiegeDefenderAdditional(): string
+CheckIfSiegeDefenderGarrison(): string
+ChecksBeforeAttack(targetArmy: Army): bool
+CreateDetachment(details: string[]): bool
+DisbandArmy()
+GetLeader(): Character
+GetLocation(): Fief
+GetSiege(): Siege
+GetTroopsEstimate(observer: Character): uint[]
+MantainArmy()
+MoveArmy(showAttrition: bool = false): bool
+UpdateArmy(): bool
```

Figure 20: Class diagram showing the Army class

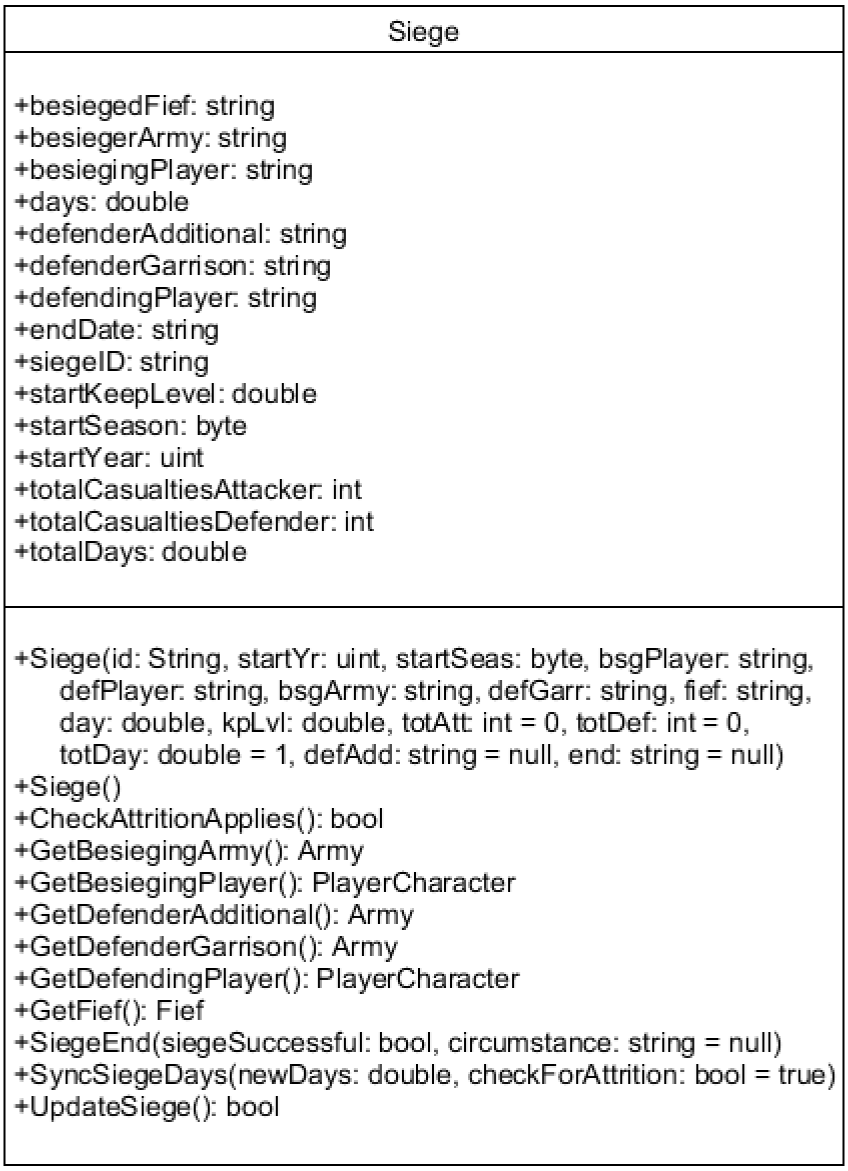
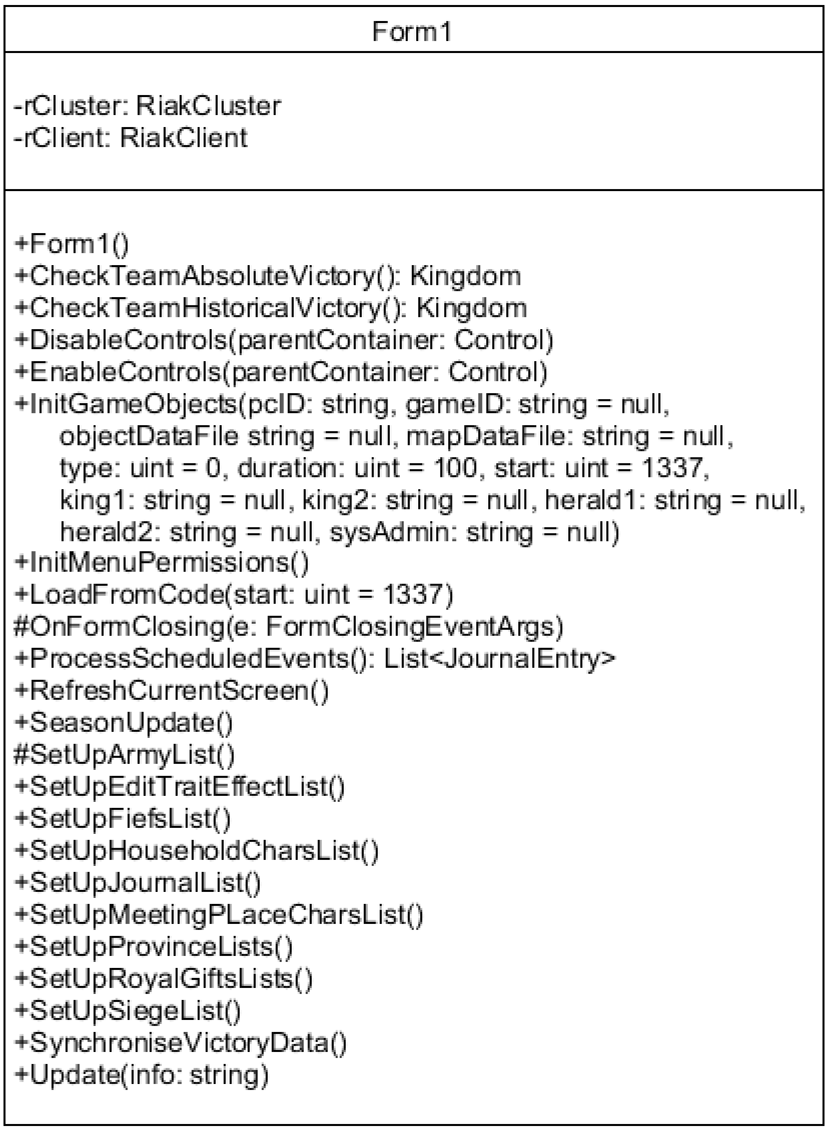

Siege
+besiegedFief: string
+besiegerArmy: string
+besiegingPlayer: string
+days: double
+defenderAdditional: string
+defenderGarrison: string
+defendingPlayer: string
+endDate: string
+siegeID: string
+startKeepLevel: double
+startSeason: byte
+startYear: uint
+totalCasualtiesAttacker: int
+totalCasualtiesDefender: int
+totalDays: double
+Siege(id: String, startYr: uint, startSeas: byte, bsgPlayer: string, defPlayer: string, bsgArmy: string, defGarr: string, fief: string, day: double, kpLvl: double, totAtt: int = 0, totDef: int = 0, totDay: double = 1, defAdd: string = null, end: string = null)
+Siege()
+CheckAttritionApplies(): bool
+GetBesiegingArmy(): Army
+GetBesiegingPlayer(): PlayerCharacter
+GetDefenderAdditional(): Army
+GetDefenderGarrison(): Army
+GetDefendingPlayer(): PlayerCharacter
+GetFief(): Fief
+SiegeEnd(siegeSuccessful: bool, circumstance: string = null)
+SyncSiegeDays(newDays: double, checkForAttrition: bool = true)
+UpdateSiege(): bool

Figure 21: Class diagram showing the Siege class



Form1

-rCluster: RiakCluster
-rClient: RiakClient

+Form1()
+CheckTeamAbsoluteVictory(): Kingdom
+CheckTeamHistoricalVictory(): Kingdom
+DisableControls(parentContainer: Control)
+EnableControls(parentContainer: Control)
+InitGameObjects(pcID: string, gameID: string = null,
    objectDataFile string = null, mapDataFile: string = null,
    type: uint = 0, duration: uint = 100, start: uint = 1337,
    king1: string = null, king2: string = null, herald1: string = null,
    herald2: string = null, sysAdmin: string = null)
+InitMenuPermissions()
+LoadFromCode(start: uint = 1337)
#OnFormClosing(e: FormClosingEventArgs)
+ProcessScheduledEvents(): List<JournalEntry>
+RefreshCurrentScreen()
+SeasonUpdate()
#SetUpArmyList()
+SetUpEditTraitEffectList()
+SetUpFiefsList()
+SetUpHouseholdCharsList()
+SetUpJournalList()
+SetUpMeetingPLaceCharsList()
+SetUpProvinceLists()
+SetUpRoyalGiftsLists()
+SetUpSiegeList()
+SynchroniseVictoryData()
+Update(info: string)

Figure 22: Class diagram showing Form1 class

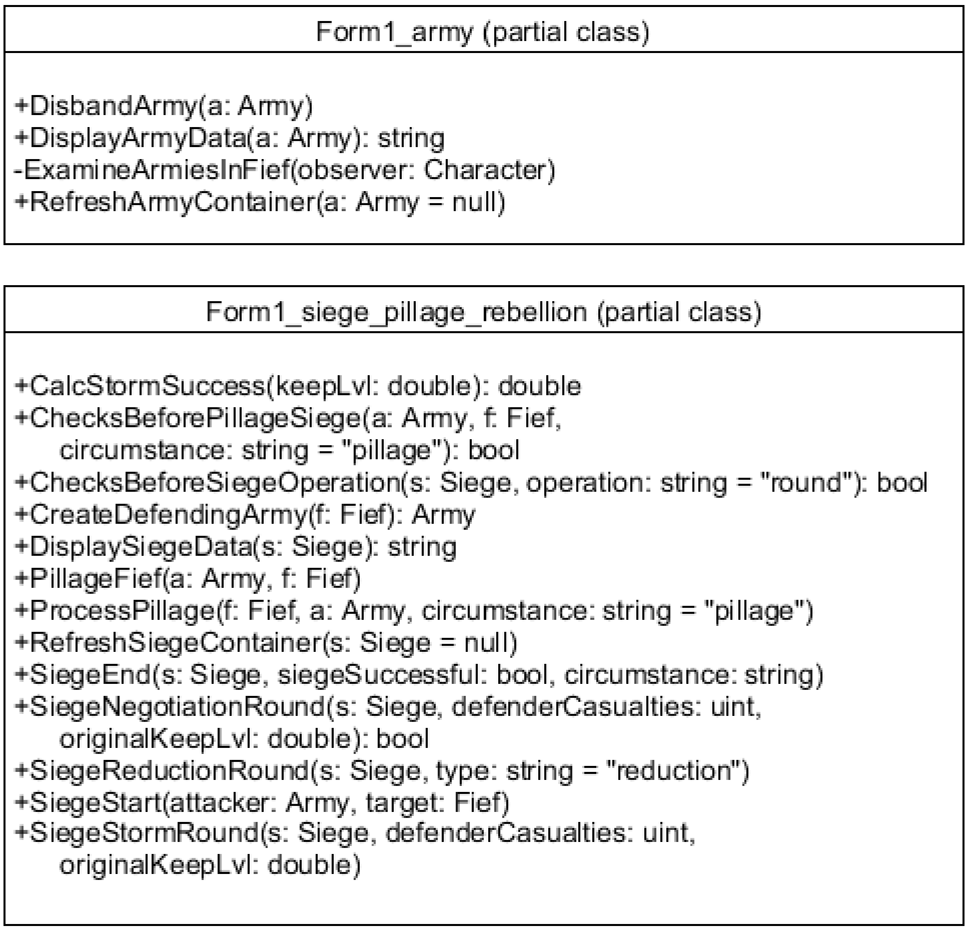
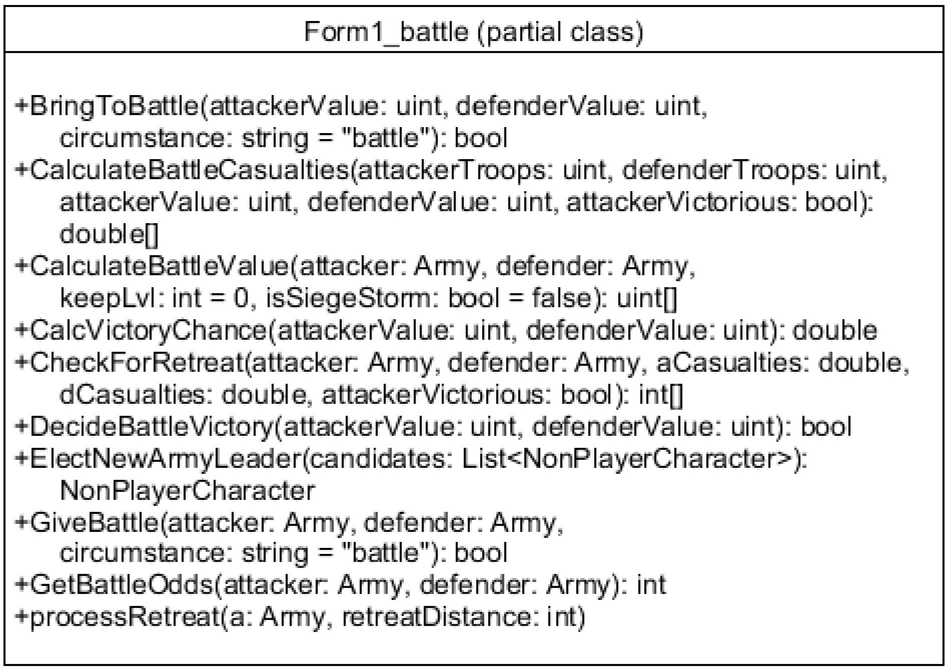

Form1_army (partial class)
+DisbandArmy(a: Army)
+DisplayArmyData(a: Army): string
-ExamineArmiesInFief(observer: Character)
+RefreshArmyContainer(a: Army = null)
Form1_siege_pillage_rebellion (partial class)
+CalcStormSuccess(keepLvl: double): double
+ChecksBeforePillageSiege(a: Army, f: Fief, circumstance: string = "pillage"): bool
+ChecksBeforeSiegeOperation(s: Siege, operation: string = "round"): bool
+CreateDefendingArmy(f: Fief): Army
+DisplaySiegeData(s: Siege): string
+PillageFief(a: Army, f: Fief)
+ProcessPillage(f: Fief, a: Army, circumstance: string = "pillage")
+RefreshSiegeContainer(s: Siege = null)
+SiegeEnd(s: Siege, siegeSuccessful: bool, circumstance: string)
+SiegeNegotiationRound(s: Siege, defenderCasualties: uint, originalKeepLvl: double): bool
+SiegeReductionRound(s: Siege, type: string = "reduction")
+SiegeStart(attacker: Army, target: Fief)
+SiegeStormRound(s: Siege, defenderCasualties: uint, originalKeepLvl: double)

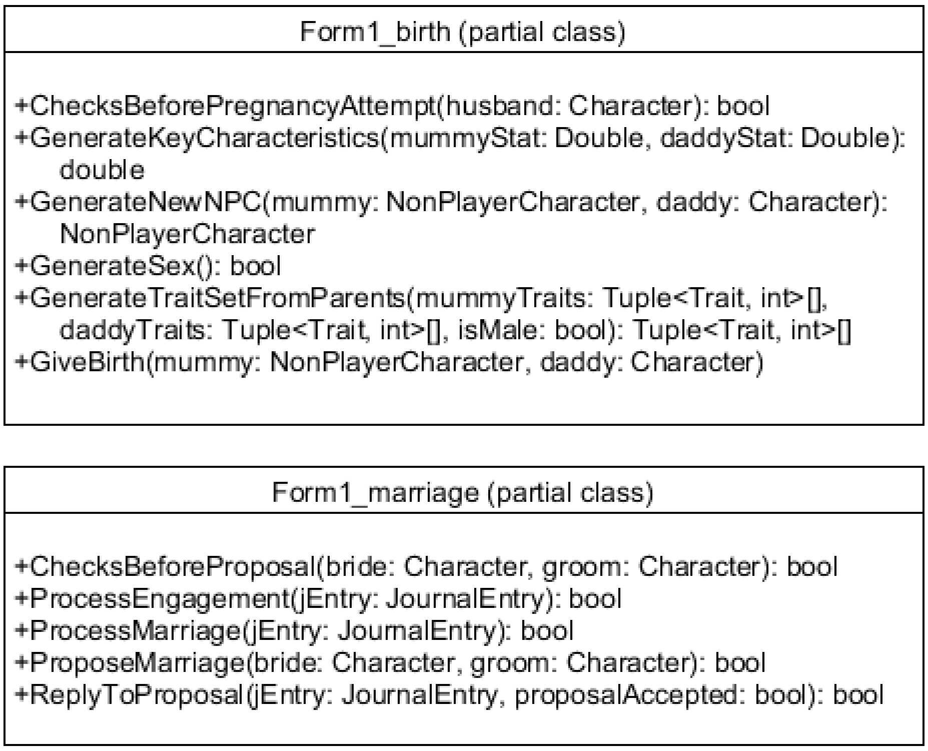
Figure 23: Class diagram showing Form1_army and Form1_siege_pillage_rebellion partial classes



```
Form1_battle (partial class)

+BringToBattle(attackerValue: uint, defenderValue: uint,
    circumstance: string = "battle"): bool
+CalculateBattleCasualties(attackerTroops: uint, defenderTroops: uint,
    attackerValue: uint, defenderValue: uint, attackerVictorious: bool):
    double[]
+CalculateBattleValue(attacker: Army, defender: Army,
    keepLvl: int = 0, isSiegeStorm: bool = false): uint[]
+CalcVictoryChance(attackerValue: uint, defenderValue: uint): double
+CheckForRetreat(attacker: Army, defender: Army, aCasualties: double,
    dCasualties: double, attackerVictorious: bool): int[]
+DecideBattleVictory(attackerValue: uint, defenderValue: uint): bool
+ElectNewArmyLeader(candidates: List<NonPlayerCharacter>):
    NonPlayerCharacter
+GiveBattle(attacker: Army, defender: Army,
    circumstance: string = "battle"): bool
+GetBattleOdds(attacker: Army, defender: Army): int
+processRetreat(a: Army, retreatDistance: int)
```

Figure 24: Class diagram showing Form1_battle partial class



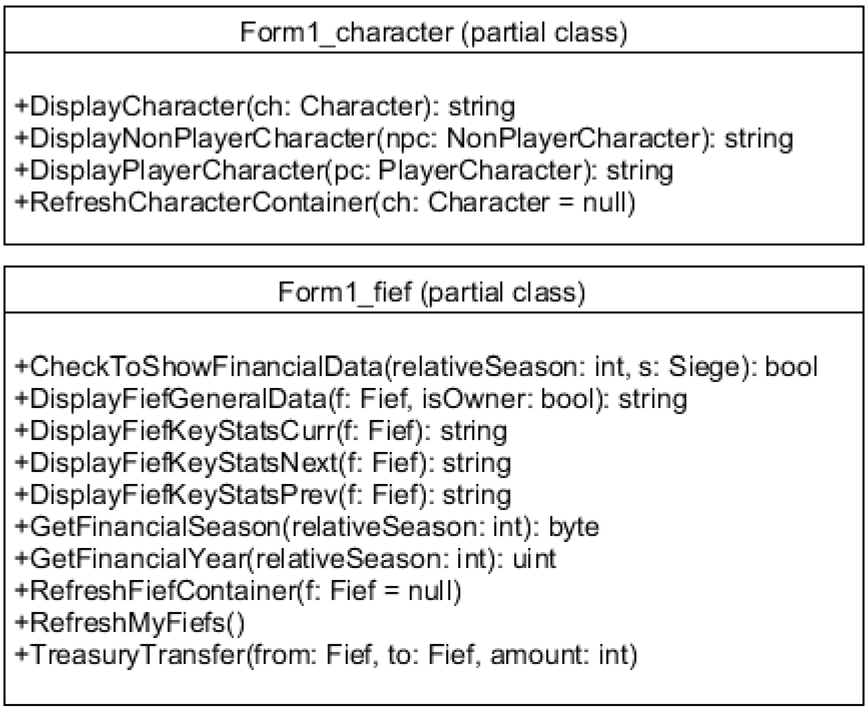
```
Form1_birth (partial class)

+ChecksBeforePregnancyAttempt(husband: Character): bool
+GenerateKeyCharacteristics(mummyStat: Double, daddyStat: Double):
    double
+GenerateNewNPC(mummy: NonPlayerCharacter, daddy: Character):
    NonPlayerCharacter
+GenerateSex(): bool
+GenerateTraitSetFromParents(mummyTraits: Tuple<Trait, int>[],
    daddyTraits: Tuple<Trait, int>[], isMale: bool): Tuple<Trait, int>[]
+GiveBirth(mummy: NonPlayerCharacter, daddy: Character)
```

```
Form1_marriage (partial class)

+ChecksBeforeProposal(bride: Character, groom: Character): bool
+ProcessEngagement(jEntry: JournalEntry): bool
+ProcessMarriage(jEntry: JournalEntry): bool
+ProposeMarriage(bride: Character, groom: Character): bool
+ReplyToProposal(jEntry: JournalEntry, proposalAccepted: bool): bool
```

Figure 25: Class diagram showing Form1_birth and Form1_marriage partial classes



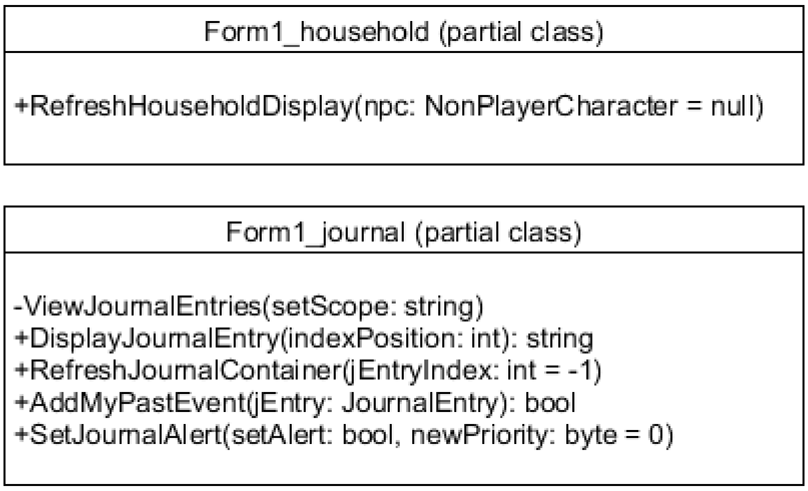
```
Form1_character (partial class)

+DisplayCharacter(ch: Character): string
+DisplayNonPlayerCharacter(npc: NonPlayerCharacter): string
+DisplayPlayerCharacter(pc: PlayerCharacter): string
+RefreshCharacterContainer(ch: Character = null)
```

```
Form1_fief (partial class)

+CheckToShowFinancialData(relativeSeason: int, s: Siege): bool
+DisplayFiefGeneralData(f: Fief, isOwner: bool): string
+DisplayFiefKeyStatsCurr(f: Fief): string
+DisplayFiefKeyStatsNext(f: Fief): string
+DisplayFiefKeyStatsPrev(f: Fief): string
+GetFinancialSeason(relativeSeason: int): byte
+GetFinancialYear(relativeSeason: int): uint
+RefreshFiefContainer(f: Fief = null)
+RefreshMyFiefs()
+TreasuryTransfer(from: Fief, to: Fief, amount: int)
```

Figure 26: Class diagram showing Form1_character and Form1_fief partial classes



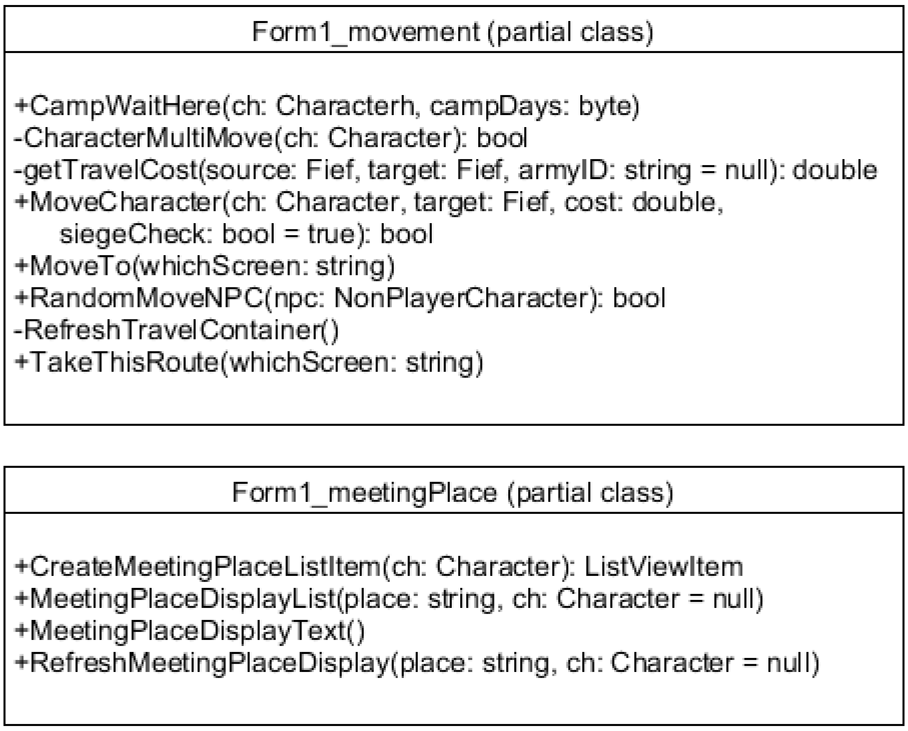
```
Form1_household (partial class)

+RefreshHouseholdDisplay(npc: NonPlayerCharacter = null)
```

```
Form1_journal (partial class)

-ViewJournalEntries(setScope: string)
+DisplayJournalEntry(indexPosition: int): string
+RefreshJournalContainer(jEntryIndex: int = -1)
+AddMyPastEvent(jEntry: JournalEntry): bool
+SetJournalAlert(setAlert: bool, newPriority: byte = 0)
```

Figure 27: Class diagram showing Form1_household and Form1_journal partial classes



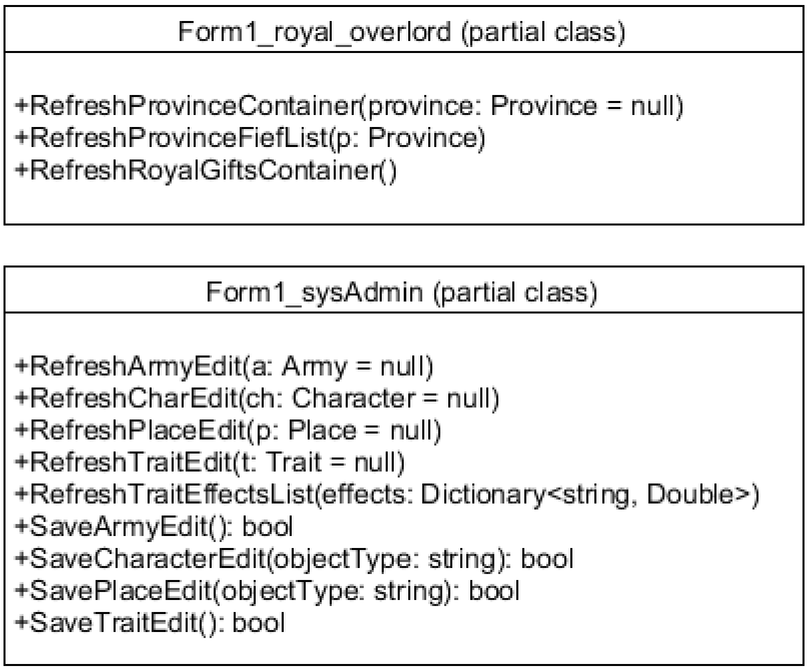
```
Form1_movement (partial class)

+CampWaitHere(ch: Characterh, campDays: byte)
-CharacterMultiMove(ch: Character): bool
-getTravelCost(source: Fief, target: Fief, armyID: string = null): double
+MoveCharacter(ch: Character, target: Fief, cost: double,
    siegeCheck: bool = true): bool
+MoveTo(whichScreen: string)
+RandomMoveNPC(npc: NonPlayerCharacter): bool
-RefreshTravelContainer()
+TakeThisRoute(whichScreen: string)
```

```
Form1_meetingPlace (partial class)

+CreateMeetingPlaceListItem(ch: Character): ListViewItem
+MeetingPlaceDisplayList(place: string, ch: Character = null)
+MeetingPlaceDisplayText()
+RefreshMeetingPlaceDisplay(place: string, ch: Character = null)
```

Figure 28: Class diagram showing Form1_movement and Form1_meetingPlace partial classes



```
Form1_royal_overlord (partial class)

+RefreshProvinceContainer(province: Province = null)
+RefreshProvinceFiefList(p: Province)
+RefreshRoyalGiftsContainer()
```

```
Form1_sysAdmin (partial class)

+RefreshArmyEdit(a: Army = null)
+RefreshCharEdit(ch: Character = null)
+RefreshPlaceEdit(p: Place = null)
+RefreshTraitEdit(t: Trait = null)
+RefreshTraitEffectsList(effects: Dictionary<string, Double>)
+SaveArmyEdit(): bool
+SaveCharacterEdit(objectType: string): bool
+SavePlaceEdit(objectType: string): bool
+SaveTraitEdit(): bool
```

Figure 29: Class diagram showing Form1_royal_overlord and Form1_sysAdmin partial classes

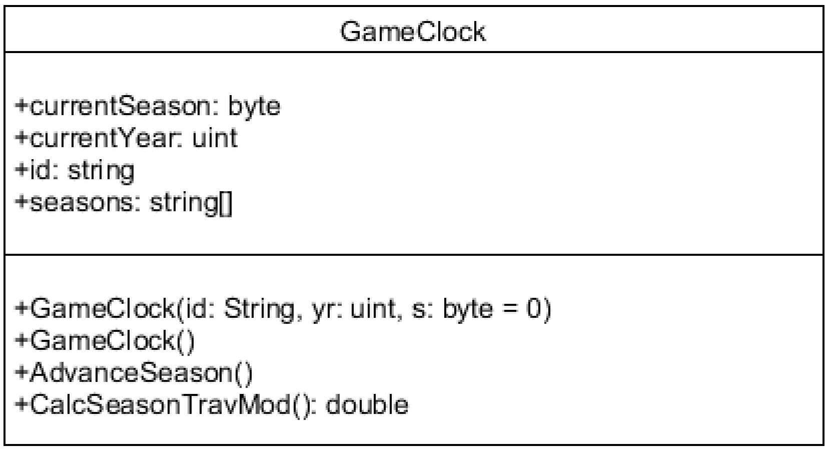


Figure 30: Class diagram showing GameClock class

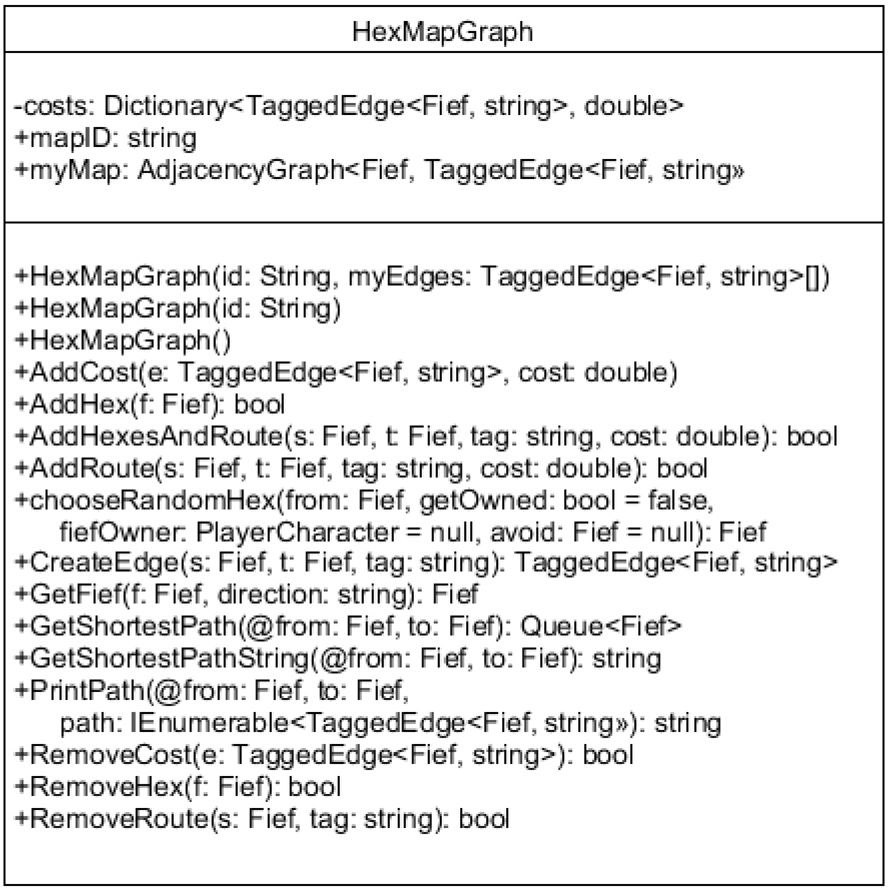

HexMapGraph
-costs: Dictionary<TaggedEdge<Fief, string>, double>
+mapID: string
+myMap: AdjacencyGraph<Fief, TaggedEdge<Fief, string»
+HexMapGraph(id: String, myEdges: TaggedEdge<Fief, string>[])
+HexMapGraph(id: String)
+HexMapGraph()
+AddCost(e: TaggedEdge<Fief, string>, cost: double)
+AddHex(f: Fief): bool
+AddHexesAndRoute(s: Fief, t: Fief, tag: string, cost: double): bool
+AddRoute(s: Fief, t: Fief, tag: string, cost: double): bool
+chooseRandomHex(from: Fief, getOwned: bool = false, fiefOwner: PlayerCharacter = null, avoid: Fief = null): Fief
+CreateEdge(s: Fief, t: Fief, tag: string): TaggedEdge<Fief, string>
+GetFief(f: Fief, direction: string): Fief
+GetShortestPath(@from: Fief, to: Fief): Queue<Fief>
+GetShortestPathString(@from: Fief, to: Fief): string
+PrintPath(@from: Fief, to: Fief, path: IEnumerable<TaggedEdge<Fief, string»): string
+RemoveCost(e: TaggedEdge<Fief, string>): bool
+RemoveHex(f: Fief): bool
+RemoveRoute(s: Fief, tag: string): bool

Figure 31: Class diagram showing HexMapGraph class

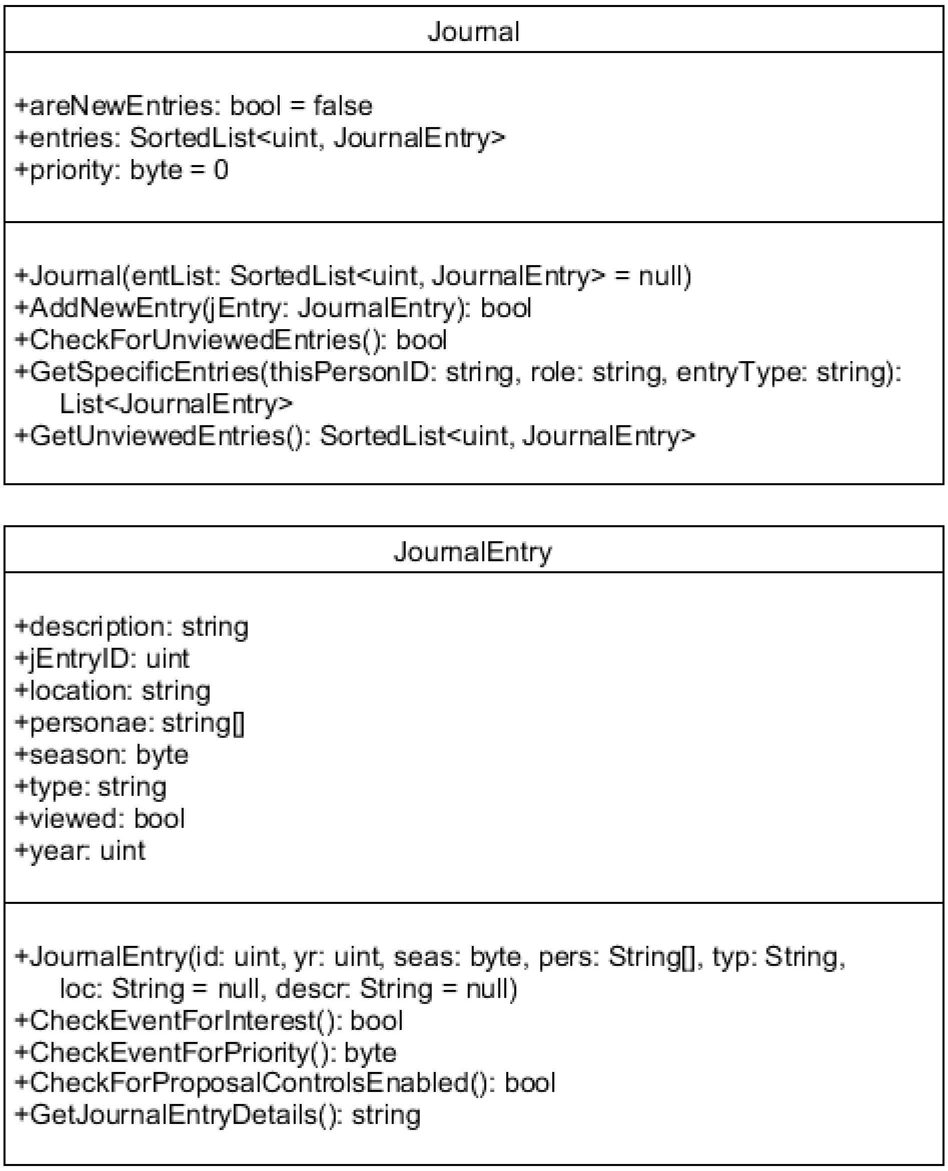

Journal
+areNewEntries: bool = false
+entries: SortedList<uint, JournalEntry>
+priority: byte = 0
+Journal(entList: SortedList<uint, JournalEntry> = null)
+AddNewEntry(jEntry: JournalEntry): bool
+CheckForUnviewedEntries(): bool
+GetSpecificEntries(thisPersonID: string, role: string, entryType: string): List<JournalEntry>
+GetUnviewedEntries(): SortedList<uint, JournalEntry>
JournalEntry
+description: string
+jEntryID: uint
+location: string
+personae: string[]
+season: byte
+type: string
+viewed: bool
+year: uint
+JournalEntry(id: uint, yr: uint, seas: byte, pers: String[], typ: String, loc: String = null, descr: String = null)
+CheckEventForInterest(): bool
+CheckEventForPriority(): byte
+CheckForProposalControlsEnabled(): bool
+GetJournalEntryDetails(): string

Figure 32: Class diagram showing Journal and JournalEntry classes

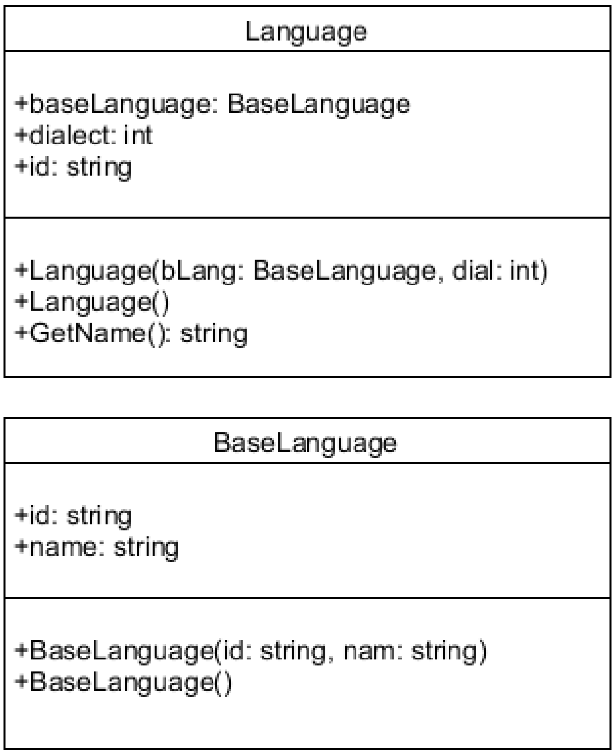


Figure 33: Class diagram showing Language and BaseLanguage classes

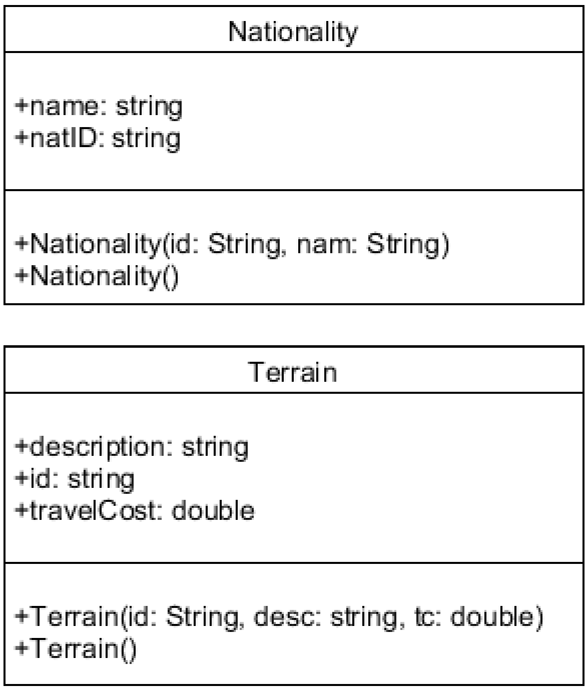


Figure 34: Class diagram showing Nationality and Terrain classes

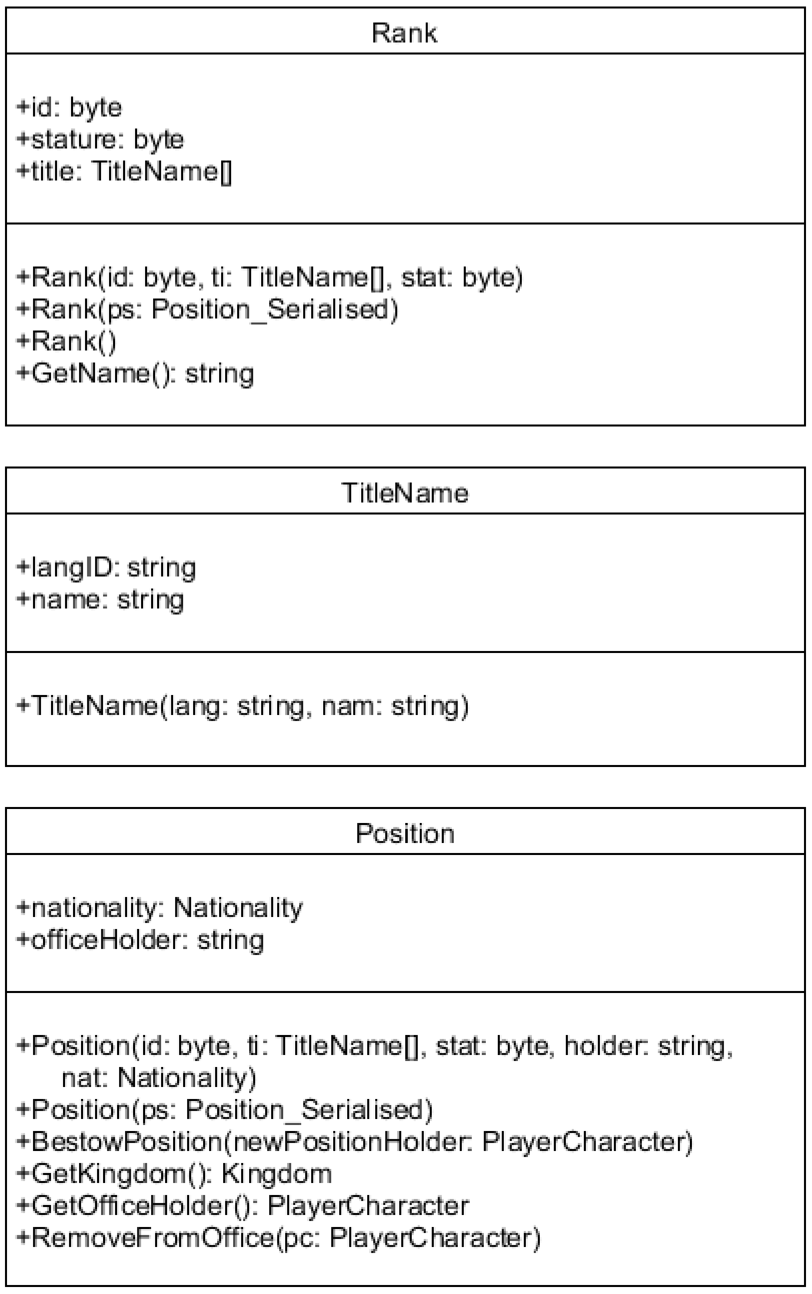
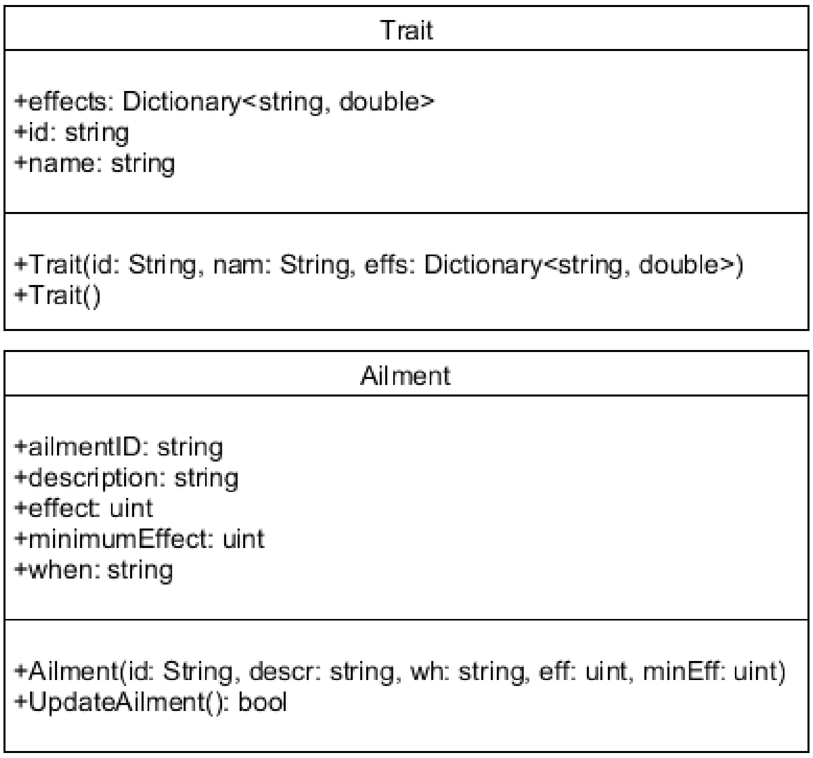

Rank
+id: byte
+stature: byte
+title: TitleName[]
+Rank(id: byte, ti: TitleName[], stat: byte)
+Rank(ps: Position_Serialised)
+Rank()
+GetName(): string
TitleName
+langID: string
+name: string
+TitleName(lang: string, nam: string)
Position
+nationality: Nationality
+officeHolder: string
+Position(id: byte, ti: TitleName[], stat: byte, holder: string, nat: Nationality)
+Position(ps: Position_Serialised)
+BestowPosition(newPositionHolder: PlayerCharacter)
+GetKingdom(): Kingdom
+GetOfficeHolder(): PlayerCharacter
+RemoveFromOffice(pc: PlayerCharacter)

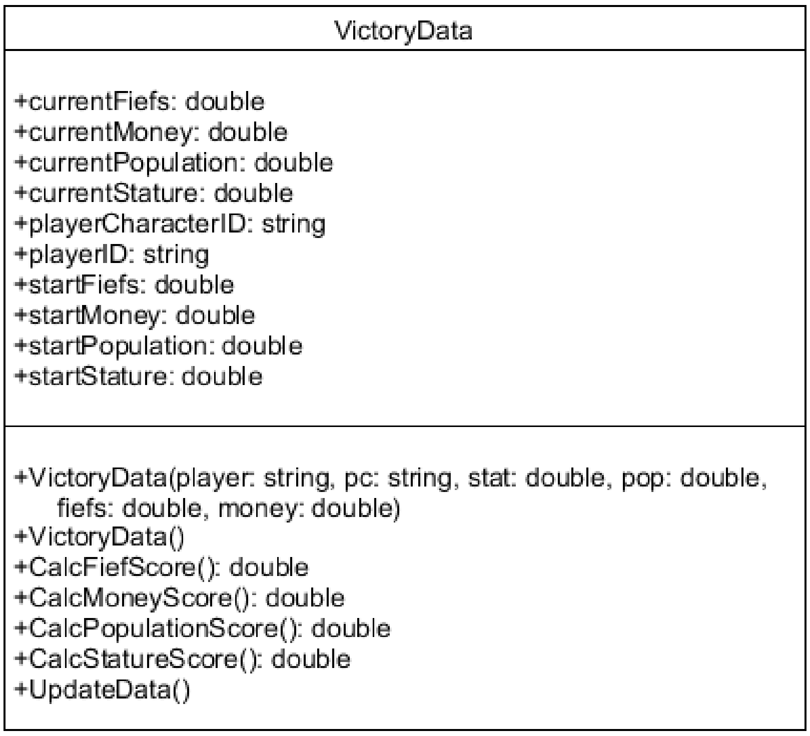
Figure 35: Class diagram showing the Rank, TitleName and Position classes



```
Trait
+effects: Dictionary<string, double>
+id: string
+name: string
+Trait(id: String, nam: String, effs: Dictionary<string, double>)
+Trait()

Ailment
+ailmentID: string
+description: string
+effect: uint
+minimumEffect: uint
+when: string
+Ailment(id: String, descr: string, wh: string, eff: uint, minEff: uint)
+UpdateAilment(): bool
```

Figure 36: Class diagram showing the Trait and Ailment classes



```
VictoryData
+currentFiefs: double
+currentMoney: double
+currentPopulation: double
+currentStature: double
+playerCharacterID: string
+playerID: string
+startFiefs: double
+startMoney: double
+startPopulation: double
+startStature: double
+VictoryData(player: string, pc: string, stat: double, pop: double,
    fiefs: double, money: double)
+VictoryData()
+CalcFiefScore(): double
+CalcMoneyScore(): double
+CalcPopulationScore(): double
+CalcStatureScore(): double
+UpdateData()
```

Figure 37: Class diagram showing the VictoryData class

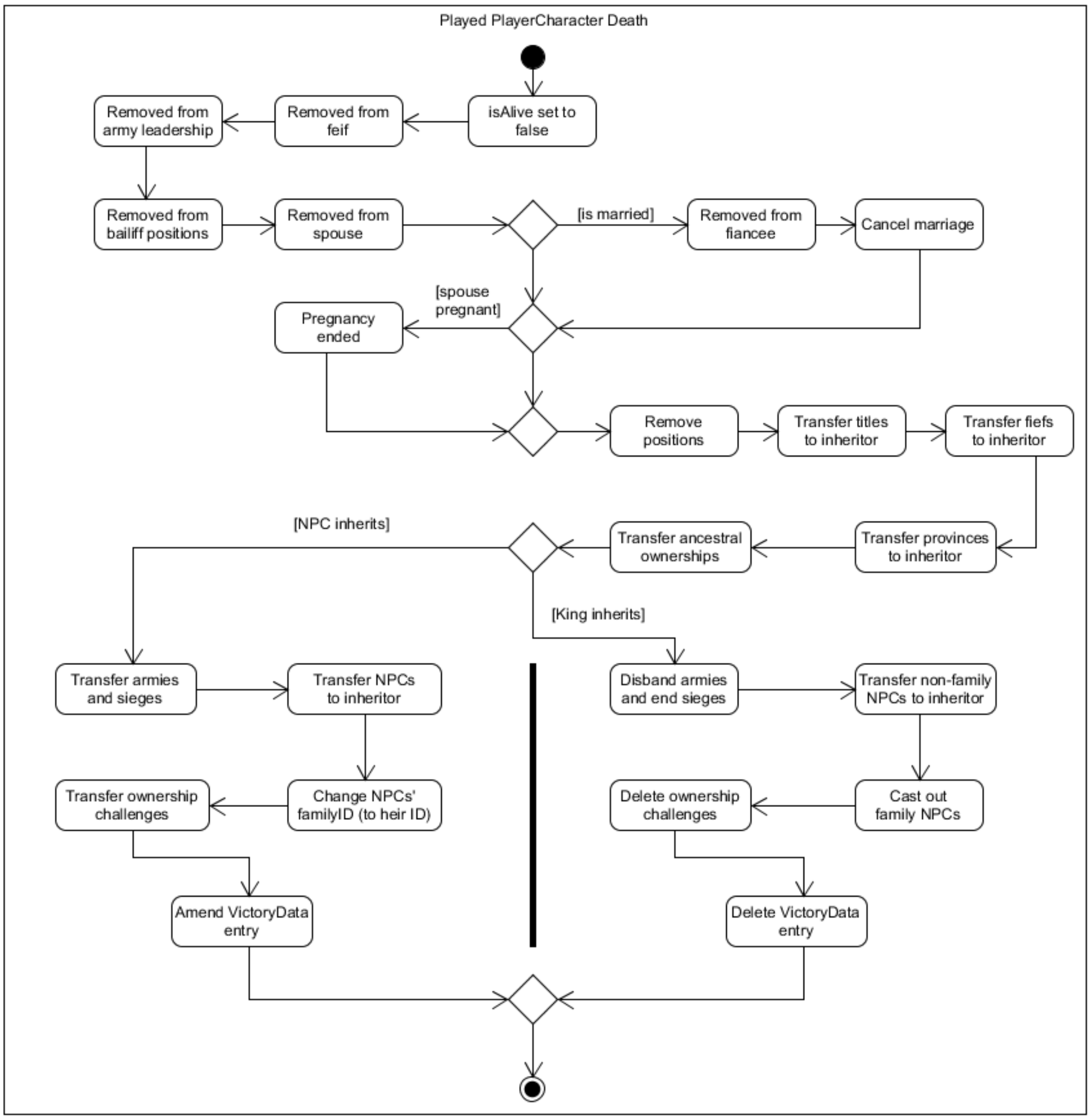

Played PlayerCharacter Death
isAlive set to false
Removed from feif
Removed from army leadership
Removed from bailiff positions
Removed from spouse
[is married]
Removed from fiancee
Cancel marriage
[spouse pregnant]
Pregnancy ended
Remove positions
Transfer titles to inheritor
Transfer fiefs to inheritor
Transfer provinces to inheritor
Transfer ancestral ownerships
[NPC inherits]
[King inherits]
Transfer armies and sieges
Transfer NPCs to inheritor
Change NPCs' familyID (to heir ID)
Transfer ownership challenges
Amend VictoryData entry
Disband armies and end sieges
Transfer non-family NPCs to inheritor
Cast out family NPCs
Delete ownership challenges
Delete VictoryData entry

Figure 38: Activity diagram showing the flow of activities involved in the death and inheritance mechanic

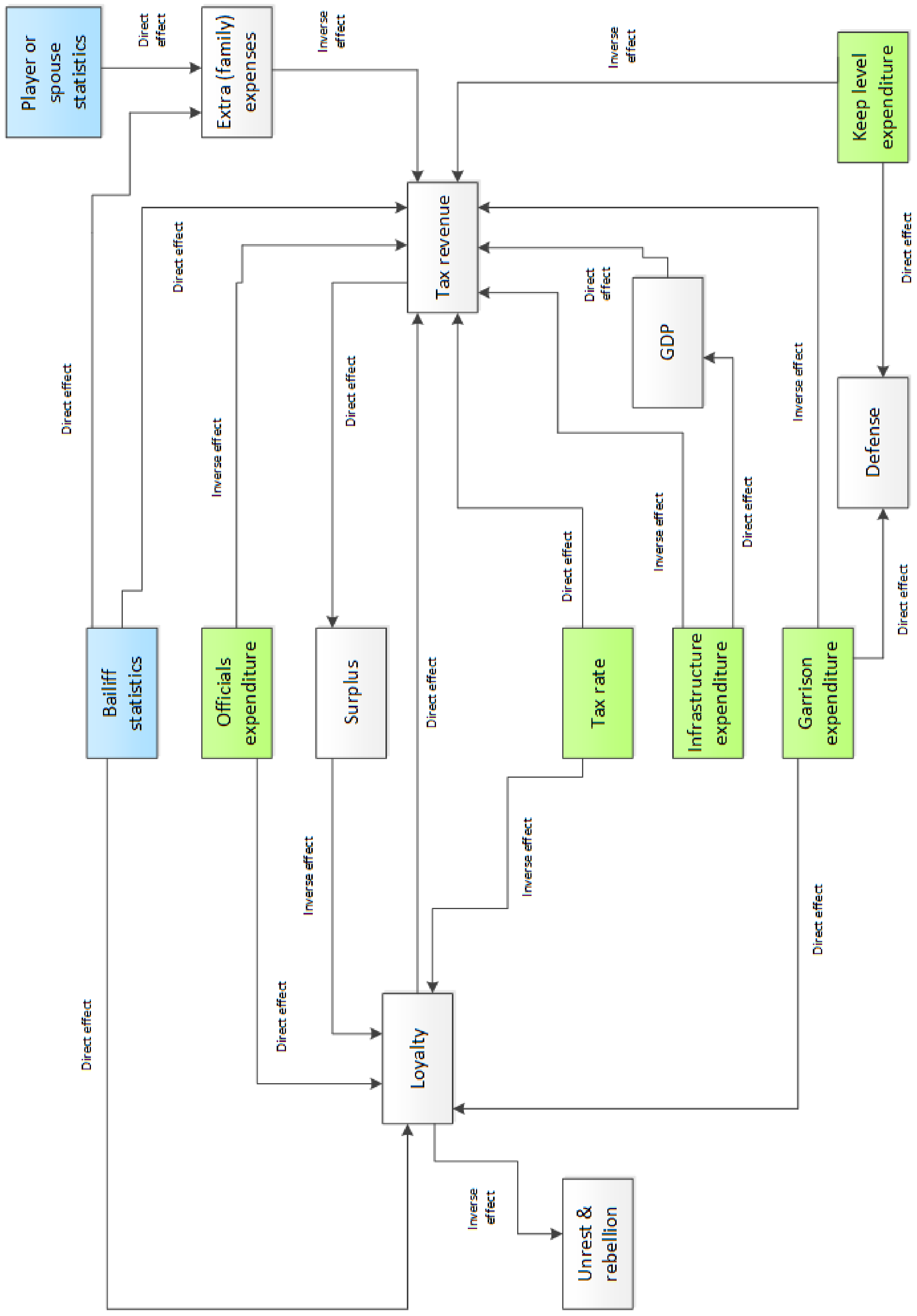


Figure 39: Entity Relationship Diagram showing the way in which the various components of the fief management mechanic influence each other

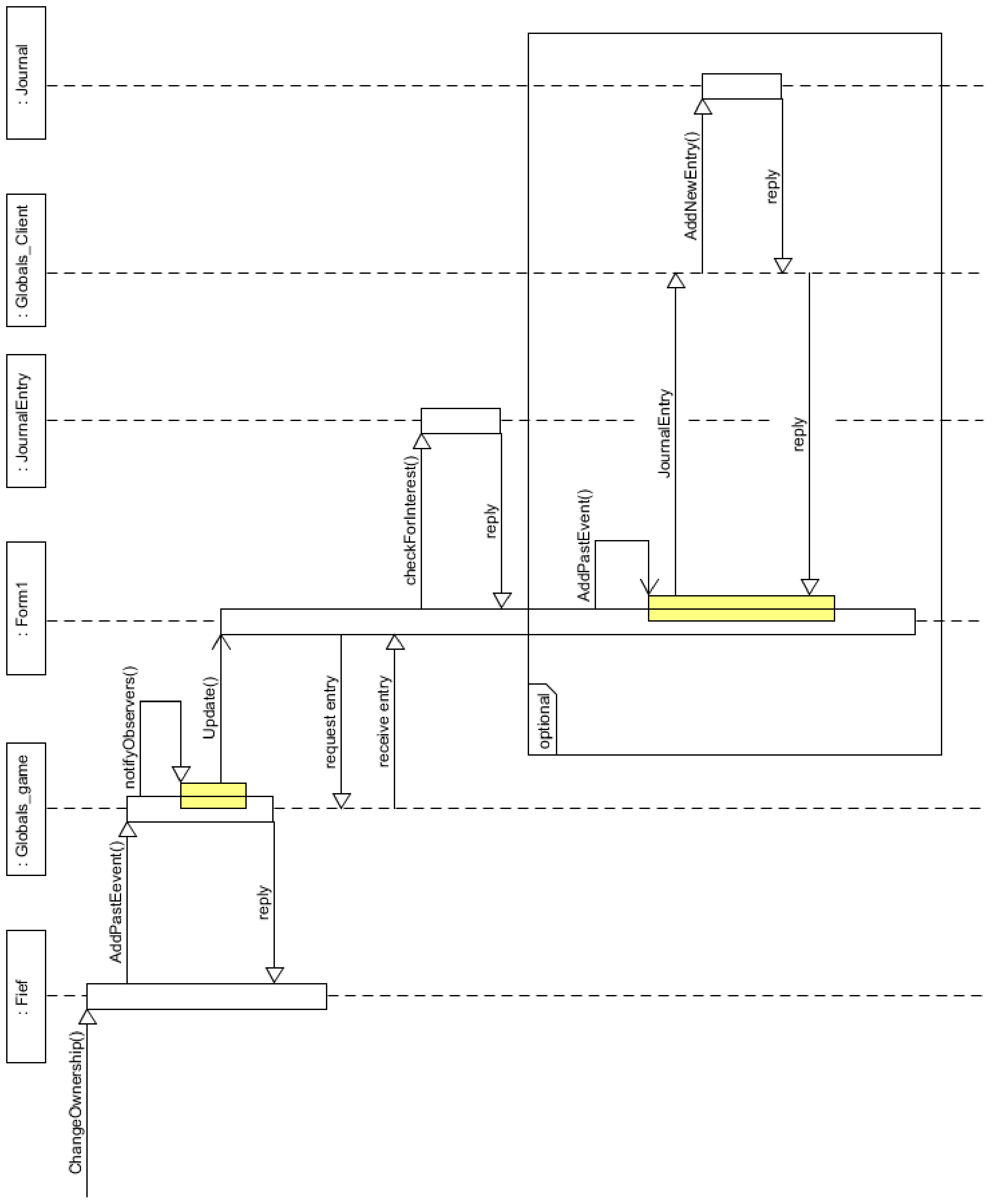


Figure 40: Sequence diagram showing the communication involved in the in-game event messaging mechanic

APPENDIX B: FORMULAS

The following formulas and game logic were used in the *JominiEngine*. Method names are supplied, where appropriate.

B.1 Armies and combat

Leadership Value (LV)

- **Method: Character.GetLeadershipValue**
- Uses: leader's combat (cbt), stature (stat), management (mgt), traits
- LV = (cbt + mgt + stat) / 3
- LV = LV + (LV * traits modifier)
- <u>Note</u>: LV of army with no leader = 4

Army combat modifier (CM)

- **Method: Form1.CalculateBattleValue**
- Uses: leader's Leadership Value (LV)
- CM = highest army LV / lowest army LV

Character combat value (CCV):

- **Method: Character.GetCombatValue**
- Uses: PC/NPC combat (cbt), health (hea), nationality (nat)
- CCV = (cbt + hea) / 2
- If (nat == 'Eng') then CCV = CCV + 5
- CCV = CCV + 5
    - The added 5 takes into account armour of PCs/NPCs
- <u>Note</u>: the above is totalled for all PCs/NPCs accompanying the army

Troop combat value (TCV):

- **Method: Army.CalculateCombatValue**
- Uses: troop combat (cbt), number (num)
- TCV = (cbt * num)
- <u>Note</u>: the above is totalled for all troop types in the army
- <u>Note</u>: in a siege round, each keep level is worth 1000 troops (foot) to the defending army

Army battle value (ABV):

- **Method: Form1.CalculateBattleValue**
- Uses: character combat value (CCV), troop combat value (TCV), army combat modifier (CM)
- ABV = CCV + TCV
- [For army with highest LV]: ABV = ABV + (ABV * CM)

Chance of bringing enemy army to battle:

- **Method: Form1.BringToBattle**
- Uses: army battle value (ABV) of both armies
- Note: this calculation is also affected by the defending army's standing orders

Condition	**Chance of battle**
Compare ABV of attacking vs defending army:	
Attacking ABV <= defending ABV	10%
Attacking ABV = (defending ABV * 2)	30%

Attacking ABV = (defending ABV * 3)	50%
Attacking ABV = (defending ABV * 4)	70%
Attacking ABV = (defending ABV * 5)	80%
Attacking CVA >= (defending CVA * 6)	90%

Table 10: Chance of bringing an enemy army to battle

Chance of winning a battle:

- **Method: Form1.CalcVictoryChance**
- Uses: army battle value (ABV) of both armies
- % chance of friendly victory = (friendly ABV / (friendly ABV + enemy ABV)) * 100

Battle casualty modifiers (CasM):

- **Method: Form1.CalculateBattleCasualties**
- Uses: army battle value (ABV) of both armies: maxABV, minABV
- Case 1: army with minABV wins
    - Victorious army CasM range = 0.1 to ((maxABV / (maxABV + minABV))) / 2
    - Losing army CasM = victorious army CasM * (0.8 to 1.2)
- Case2: army with maxABV wins
    - Victorious army CasM = (1 + ((minBV * minBV) / (maxBV * maxBV))) / 2
    - Losing army CasM range = 0.1 to (maxABV / (maxABV + minABV))
- Application of CasM: troop number = troop number – (troop number * CasM)

Retreat from battle:

- **Method: Form1.CheckForRetreat and Form1.ProcessRetreat**
- An army will retreat if:
    - Specified by standing orders and not brought to battle
    - It is the loser in battle and suffers >= 20% casualties
    - It has unsuccessfully attempted to quell a rebellion
- Retreat length:
    - If due to casualties, retreat is random 1-2 hexes
    - Otherwise, retreat is 1 hex
- <u>Note</u>: The retreating army will try to retreat into friendly fiefs

Chance of siege storm/negotiation success:

- **Method: Form1.CalcVictoryChance and Form1.SiegeNegotiationRound**
- Uses: army battle value (ABV) of both armies
- % chance of storm success = (attacker ABV / (attacker ABV + defender ABV)) * 100
- % chance of negotiated success = as above / 2

Siege effects (casualties and keep level):

- **Method: Form1.SiegeStormRound and Form1.SiegeReductionRound**
- Uses: fief keep level (keepLvl), army battle value (ABV) of both armies
- For each <u>non-storm</u> siege round:
    - keepLvl = keepLvl * 0.92
    - Defending casualty modifier = 0.01
    - % chance of defender leader injury: 1
- For each <u>storm</u> siege round (in <u>addition</u> to above):
    - keepLvl = keepLvl * 1 - (0.1 + 0.015 * (attacker ABV / defender ABV))
    - Defender casualty modifier (CasM) = 0.01 * (attacker ABV / defender ABV)
    - Attacker CasM = 0.01 * (defender ABV / attacker ABV)

- Case 1: storm success
  - Attacker CasM += 0.005 * keepLvl
- Case 2: storm failure
  - Attacker CasM += 0.01 * keepLvl

Recruitment:

- **Method: Fief.CallUpTroops**
- Uses: fief population (pop), status (stat)
- Maximum troops (maxTrp) available in fief = pop * 0.05
- Number recruited (numRaised): maxTrp * (random between 0-1) [both can be specified]
- If stat = 'U': numRaised = numRaised / 2
- Cost:
  - Non-ancestral fiefs: 2,000 per man
  - Ancestral fiefs: 500 per man
  - Funds come from home treasury
- Conditions:
  - Fief must be owned by player
  - Can only recruit once per season in a fief
  - If fief speaks different major language, can only recruit if fief loyalty >= 7
  - Cannot raise troops if fief in rebellion

Size of defending army (garrison + militia) in pillage/siege:

- **Method: Form1.CreateDefendingArmy**
- Uses: fief population (pop), garrison spend (gSpend)
- Maximum troops (maxTrp) available in fief = pop * 0.05
- Militia size = maxTrp * (random between 0.33 and 0.66)
- Garrison size = gSpend / 1000
- Defending army size = garrison + militia
- Note: national recruitment ratios determine exact numbers of each troop type
- Note: garrison consists of entirely professional troops (no rabble)

Chance of defending (fief) army giving battle prior to pillage:

- **Method: Form1.BringToBattle**
- Uses: army battle value (ABV) of both armies
- Note: chances are less than with field battle to reflect the unprepared state of defending forces

Condition	**Chance of battle**
Defending ABV <= attacking ABV	10%
Defending ABV = (attacking ABV * 2)	20%
Defending ABV = (attacking ABV * 3)	30%
Defending ABV = (attacking ABV * 4)	40%
Defending ABV = (attacking ABV * 5)	50%
Defending ABV >= (attacking ABV * 6)	60%

Table 11: Chance of bringing an enemy army to battle prior to pillage

Chance of quelling rebellion:

- **Method: Fief.Quell_checkSuccess**

- Uses: fief population (pop), fief's ancestral owner (ancOwner), army size (aSize), army leader's leadership value (LV), army owner (armOwner), army owner stature (stat)
- Chance of success (quellChance) = aSize / (pop / 1000)
- quellChance = quellChance + LV
- If ancOwner = armOwner: quellChance = quellChance + (quellChance * (stat * 2.22))
- <u>Note</u>: maximum quellChance = 99

Pillage (or successful siege storm) results:

- **Method: Form1.ProcessPillage**
- Uses: fief population (pop), GDP (gdp), loyalty (loy), treasury (treas), fields (fld), industry (ind), pillaging army size, pillaging army leader stature (stat), no. of days pillaging
- Pillage multiplier (pillMult) = pillaging army size / (pop / 1000)
- Pop = pop – (pop * ((0.007 * pillMult) / 100) [min 0.01, max 0.2]
- Treas = treas - (treas * ((0.2 * pillMult) / 100) [min 0.01, max 0.8]
- Loy = loy - (loy * ((0.33 * pillMult) / 100) [min 0.01, max 0.2]
- Fld = fld - (fld * ((0.01 * pillMult) / 100) [min 0.01, max 0.2]
- Ind = ind - (ind * ((0.01 * pillMult) / 100) [min 0.01, max 0.2]
- Money pillaged (monPill) = gdp * ((0.032 * pillMult) / 100) [min 0.01, max 0.5]
  - For each day pillaging > 7: monPill = monPill + (monPill * 0.05)
  - 30% chance (jackpot!): monPill = monPill * 3-10 (random)
- Army leader money pillaged = monPill * (0.05 * stat) ProcessPillage

General siege effects:

- Fief management:
  - All expenditure reduced to 0 except for garrison
  - Tax rate remains unchanged but is reduced to 0 for the purposes of calculating income (i.e. income = 0).
  - Expenses for family members in besieged keep are reduced by 50%.
- Armies:
  - Defending armies cannot move (restriction applies to leader, as this is how armies move).
  - Players can't recruit from a fief under siege.
  - Players can't pillage a fief under siege.
- Household:
  - Marriages postponed if one of the couple is trapped in a besieged keep, and the other isn't.
  - The above also applies to pregnancy attempts.

PC/NPC injury in battle or siege (storm only):

- **Method: Character.CalculateCombatInjury**
- Uses casualty modifier of friendly army (CasM), character combat (cbt), character health (hea)
- Base % chance of injury (iChance) = CasM * 100
- iChance = iChance + (5 - cbt) [min 1, max 80]
- Effect of injury: Ailment with 1-5 effect (random)
  - If (hea – ailment effect) <= 0: character death
- If (ailment effect = 5): ailment minimumEffect = 1

Army attrition:

- **Method: Army.CalcAttrition**

- Uses fief population (pop), army size (aSize), leader's management (man) and stature (stat)
- % chance of occurrence (attChance) = ((aSize / pop) * 100) – (stat + man / 2)
    - If winter/spring: attChance = attChance + 20
    - Minimum attChance = 10
- Casualty modifier (CasM) = (aSize / pop) / 10
    - If winter/spring: CasM = CasM * 3
- When attrition occurs:
    - Every time army moves into new fief (using new fief's pop)
    - For every 7 days in place:
        - left at end of season
        - camped
        - spent organising transfer
        - etc.
    - Siege:
        - Besieging army: yes
        - Defending army: If (no. of days in siege > (bailiff man * 60))

Army maintenance:

- **Method: Army.MantainArmy**
- Cost: 500 per man per season
- Maintained armies don't suffer attrition

Standing orders:

- Based on aggression level and Combat Odds value.
- Aggression:
    - 0: If outside keep, will attempt to retreat from the fief if attacked. If inside keep, aggression changes to 1, to allow participation in siege defence.
    - 1: If outside keep, will check Combat Odds; if enemy vs friendly odds are <= specified level, will defend when attacked; if not, will attempt to retreat. Will NOT attempt to proactively prevent siege/pillage. If inside keep, will check Combat Odds; if friendly vs enemy odds are >= specified level, will sally and attack.
    - 2: If outside keep, will not allow siege/pillage until driven from field and will always give battle when attacked. If inside keep, will check Combat Odds; if friendly vs enemy odds are >= specified level, will sally and attack.
- Combat Odds: Required odds before will attempt an attack (or defence, if aggression is 1). I.e. 1 = even odds, 2 = 2-to-1 odds, etc.
- <u>Note</u>: odds are always calculated as attacker to defender
- <u>Note</u>: If army leaderless, will revert to aggression of 1 and odds of 6.

Estimating enemy army size:

- **Method: Character.GetEstimateVariance**
- Uses: observer's leadership value (LV), army size (aSize)
- Base estimate variance (EV) = 0.05
- EV = base EV + ((10 – LV) * 0.05)
- Estimate = aSize +/- (aSize * EV)

B.2 Fiefs

Default settings if no bailiff:

- Average characteristics of 3

Calculation of GDP:

- **Method: Fief.CalcNewGDP**
- Uses: fief fileds (fld), industry (ind), population (pop)
- GDP = ((fields x 8997) + (industry x pop)

Rate of population increase:

- **Method: Fief.CalcNewPopulation**
- Uses: fief population (pop)
- Pop = pop + (pop * 0.005)
- <u>Note</u>: this is performed once per season (2% per year)

Calculation of effect of tax rate on loyalty:

- **Method: Fief.CalcNewLoyalty**
- Uses: fief current tax rate (tx), next season's tax rate (txNxt), loyalty (loy)
- loy = loy + loy * ((txNxt - tx) / 100) * -1)

Effect of bailiff stats on fief loyalty:

- **Method: Fief.CalcBaseFiefLoyMod**
- Uses: bailiff language (bLang), management (man), stature (stat), fief language (fLang), loyalty (loy)
- Bailiff loyalty modifier (loyMod) = ((man + stat / 2) -1)
- If bLang != fLang: loyMod = loyMod – 3
- loyMod = loyMod * 0.0125
- Applied: loy = loy + (loy * loyMod)

Effect of bailiff on income:

- **Method: Fief.CalcFiefIncMod**
- Uses bailiff management (man), fief income (inc)
- Bailiff income modifier (incMod) = man - 1 * 0.025
- Applied: inc = inc + (inc * incMod)

Effect of officials expenditure on loyalty & income:

- **Method: Fief.CalcOffLoyMod and Fief.CalcOffIncMod**
- Uses: fief population (pop), loyalty (loy), income (inc), officials expenditure (offSpend)
- Max offSpend per 1 pop = 4
- 'Neutral' offSpend per 1 pop = 2
- Loyalty modifier (loyMod) = ((offSpend - (pop * 2)) / (pop * 2)) / 10
- Income modifier (incMod) = ((offSpend - (pop * 2)) / (pop * 2)) / 10
- Applied: loy = loy + (loy * loyMod)
- Applied: inc = inc + (inc * incMod)
- <u>Note</u>: Effects loy & inc in range of -10% to 10%, depending on offSpend

Effect of garrison expenditure on loyalty:

- **Method: Fief.CalcGarrLoyMod**
- Uses: fief population (pop), loyalty (loy), garrison expenditure (gSpend)
- Max gSpend per 1 pop = 14
- 'Neutral' gSpend per 1 pop = 7
- Loyalty modifier (loyMod) = ((gSpend - (pop * 7)) / (pop * 7)) / 10
- Applied: loy = loy + (loy * loyMod)

- Note: Effects loy in range of -10% to 10%, depending on gSpend

Effect of bailiff (owner/owner's spouse in home fief) on family expenses:

- **Method: Fief.CalcFamExpenseMod**
- Uses: character management (man), fief family expenses (famExp)
- Family expenses modifier (famExpMod) = (((man - 1) * 2.5) / 100) * -1
- Applied: famExp = famExp + (famExp * famExpMod)
- Note: Range of decrease 0%-20%

Effect of unrest or rebellion on income:

- **Method: Fief.CalcStatusIncmMod**
- Uses: fief income (inc), status (stat)
- If stat = 'U': Income modifier (incMod) = 0.5
- If stat = 'R': incMod = 0
- Applied: inc = inc * incMod

Calculation of chance of rebellion or unrest:

- **Method: Fief.CheckFiefStatus**
- Uses: fief tax rate (tx), income (inc), surplus (surp), loyalty (loy)
- Method 1: If tx > 20 and surp > (inc * 0.1), then % chance of rebellion = tx - 20
- Method 2: If loy > 3 and <=4, then 10% chance for unrest and 2% chance for rebellion
  - If loy > 2 and <=3, then 30% chance unrest and 14% rebellion
  - If loy > 1 and <=2, then 50% chance unrest and 26% rebellion
  - If loy > 0 and <=1, then 70% chance unrest and 38% rebellion
  - If loy 0, then 90% chance unrest and 50% rebellion

Effect of surplus size on fief loyalty:

- **Method: Fief.CalcNewLoyalty**
- Uses: fief income (inc), surplus (surp), loyalty (loy)
- loy = loy – (surp/ inc)

Effect of infrastructure expenditure on field level:

- **Method: Fief.CalcNewFieldLevel**
- Uses: fief fields (fld), infrastructure expenditure (infSpend)
- fld = fld + (infSpend / 500000)
- Note: If infSpend = 0: fld = fld – (fld / 100)
- Note: Max infSpend per 1 pop = 6

Effect of infrastructure expenditure on industry level:

- **Method: Fief.CalcNewIndustryLevel**
- Uses: fief industry (ind), infrastructure expenditure (infSpend)
- ind = ind + (infSpend / 1500000)
- Note: If infSpend = 0: ind = ind – (ind / 100)
- Note: Max infSpend per 1 pop = 6

Effect of keep expenditure on keep level:

- **Method: Fief.CalcNewKeepLevel**
- Uses: fief keep level (keepLvl), keep expenditure (kpSpend)
- keepLvl = keepLvl + (kpSpend / 400000)
- Note: If kpSpend = 0: keepLvl = keepLvl – 0.15

- Note: Max spend per 1 pop = 13

B.3 Characters

Traits:

- **Method: Character.CalcTraitEffect**
- Applied:
  - Trait base modifiers are a multiplier effect between 0.05 and 0.4 (+ or -).
  - The character's trait level is applied to the base modifier to derive the final modifier:
    - Trait modifier = (level * 0.111) * base modifier
    - Example: a character with a level 6 'Command' trait, would have a 'battle' modifier of 0.2664: (6 * 0.111) * 0.4
  - Trait modifiers are applied to the combination of characteristics used in that particular operation; e.g. the modifier would be applied to the leadership value (LV) rather than the separate combat, management and stature characteristics.

Note: **Red shows negative effects** and **green shows positive effects**

Command:

- **+ Battle orders (LV in battle) (+0.4)**
- **+ Siege success (LV in siege) (+0.4)**
- **+ NPC hire (+0.2)**

Chivalry

- **+ Family expenses (+0.2)**
- **+ Fief expenses (total expenses – family expenses) (+0.1)**
- **+ Fief loyalty (+0.2)**
- **+ NPC hire (+0.1)**
- **+ Siege success (LV in siege) (+0.1)**

Abrasiveness

- **+ Battle orders (LV in battle) (+0.15)**
- **+ Death probability (+0.05)**
- **- Fief expenses (total expenses – family expenses) (-0.05)**
- **+ Family expenses (+0.05)**
- **- Time efficiency (days per turn) (+0.05)**
- **- Siege success (LV in siege) (-0.1)**

Accountancy:

- **+ Time efficiency (days per turn) (+0.1)**
- **- Fief expenses (total expenses – family expenses) (-0.2)**
- **- Family expenses (-0.2)**
- **- Fief loyalty (-0.05)**

Stupidity

- **- Battle orders (LV in battle) (-0.4)**
- **+ Death probability (+0.05)**
- **+ Family expenses (+0.2)**
- **+ Fief expenses (total expenses – family expenses) (+0.2)**

- **- Fief loyalty (-0.1)**
- **- NPC hire (-0.1)**
- **- Time efficiency (days per turn) (-0.1)**
- **- Siege success (LV in siege) (-0.4)**

Robust

- **+ Virility (+0.2)**
- **+ NPC hire (+0.05)**
- **+ Fief loyalty (+0.05)**
- **- Death probability (-0.2)**

Pious

- **+ Fief loyalty (+0.1)**
- **+ NPC hire (+0.1)**
- **- Time efficiency (days per turn) (-0.1)**
- **- Virility (-0.2)**

Hiring NPCs:

- **Method: NonPlayerCharacter.CalcSalary_BaseOnTraits and CalcSalary_BaseOnCurrent**
- Uses NPC management (man), combat (cbt), stature (npcStat), and current salary (currSal); PC stature (pcStat); employer stature (emplStat)
- Part 1: Based on skills
  - Base salary (baseSal) = 1500
  - NPC fief management rating (fiefMgt) = (man + npcStat) / 2
  - Trait effects: fiefMgt = fiefMgt + (fiefMgt * (fiefLoyalty + (-1 * fiefExpenses)))
  - NPC leadership value (LV) = man + npcStat + cbt / 3
  - Trait effects: LV = LV + (LV * (battle + siege))
  - Final salary (finSal) = baseSal * (highest of fiefMgt or LV)
  - Flexibility bonus: finSal = finSal + (baseSal * (lowest of fiefMgt and LV / 2)
- Part 2: Based on currSal (if employed)
  - Final salary (finSal) = (currSal + (currSal * 0.05))
  - finSal = finSal + (finSal * 0.11)
- Use highest finSal derived from both methods
- Get PC/employer stature modifier (statMod)
  - If pcStat > 4: PC modifier (pcMod) = (pcStat – 4) * 0.04
  - If emplStat > 4: employer modifier (emplMod) = ((emplStat – 4) * 0.04) * -1)
  - statMod = 1 – (pcMod + emplMod)
- finSal = finSal * statMod
- Apply PC npcHire traits effect to chance of offer being accepted (not salary)

Calculation of stature:

- **Method: Character.CalculateStature**
- Uses: character stature (stat)
- Base stature:
  - Rank
    - Rank 1-3 (Popes and Kings): stat = (stat + 6)
    - Rank 4-7 (Prince-Bishops and Dukes)): stat = (stat + 5)
    - Rank 8-9 (Princes)): stat = (stat + 4)
    - Rank 10-12 (Marquis and Counts)): stat = (stat + 3)
    - Rank 13-15 (Viscount and Barons)): stat = (stat + 2)

        - Rank 16-17 (just about everyone else)): stat = (stat + 1)
    - Age
        - => 61: stat = (stat + 5)
        - 51-60: stat = (stat + 4)
        - 41-50: stat = (stat + 3)
        - 31-40: stat = (stat + 2)
        - 21-30: stat = (stat + 1)
        - 11-20: stat = (stat + 0.5)
    - Sex:
        - If female: stat = (stat - 6)
- Stature modifiers (applied to *Character.statureModifer*):
    - If win battle: stat = stat + (0.8 *(enemy army size / 10000))
    - If lose battle: stat = stat - (0.5 *(enemy army size / 10000))
    - If win siege by negotiation: stat = stat + (0.2 *(fief pop / 10000))
    - If win siege by storm: stat = stat + (0.1 *(fief pop / 10000))
    - If storm keep during siege and lose: stat = stat - (0.2 *(fief pop / 10000))
    - If daughters, granddaughters, etc. marry into family of higher rank: stat = stat + (0.4 *(higher rank – your rank))
    - If fief rebels: stat = stat - 0.1
    - If pillage fief (of same base language): stat = stat - 0.2
    - If pillage fief (of same base language & dialect): stat = stat - 0.3

Chance of pregnancy:

- **Method: Character.GetSpousePregnant**
- Uses wife's (wAge) and virility (wVir); husband's virility (hVir)
- Base % pregnancy chance (pregChance):
    - If wAge = 14-17: pregChance = 8%
    - If wAge = 18-24: pregChance = 10%
    - If wAge = 25-29: pregChance = 8%
    - If wAge = 30-34: pregChance = 6%
    - If wAge = 35-39: pregChance = 5%
    - If wAge = 40-44: pregChance = 4%
    - If wAge = 45-49: pregChance = 2%
    - If wAge = 50-55: pregChance = 1%
- If wAge <= 55: pregChance = pregChance + pregChance * (((hVir + wVir) / 2) -5) / 10

Calculation of health:

- **Method: Character.CalculateHealth**
- Uses character maximumHealth (maxHea) and age
- Age modifier (ageMod):
    - If age = 0: ageMod = 0.25
    - If age = 1-4: ageMod = 0.5
    - If age = 5-9: ageMod = 0.8
    - If age = 10-19: ageMod = 0.9
    - If age = 20-34: ageMod = 1.0
    - If age = 35-39: ageMod = 0.95
    - If age = 40-44: ageMod = 0.9
    - If age = 45-49: ageMod = 0.85
    - If age = 50-54: ageMod = 0.75
    - If age = 55-59: ageMod = 0.65
    - If age = 60-69: ageMod = 0.55

    - If age > 70: ageMod = 0.35
- Health = (maxHea * ageMod)

Base chance of death:

- **Method: Character.CheckForDeath**
- Uses character health (hea) and sex
- Death modifier (deathMod) for Men: 2.8
- deathMod for Women: 2.5
- Base chance of death = (10 – hea) * deathMod

Chance of death in childbirth:

- **Method: Character.CheckForDeath**
- Uses character base death chance (baseChance)
- % chance of baby death (babyChance) = (baseChance * 1.5)
- % chance of mother death (mumChance) = (baseChance * 1.5)
    - If baby stillborn: mumChance = (baseChance * 2)

Calculation of child characteristics:

- **Method: Form1.GenerateKeyCharacteristics**
- % chance for each characteristic:
    - 35% = parental avg
    - 17.5% = (parental avg - 1)
    - 17.5% = (parental avg + 1)
    - 10% = (parental avg - 2)
    - 10% = (parental avg + 2)
    - 5% = (parental avg - 3)
    - 5% = (parental avg + 3)

Calculation of family expenses:

- **Method: Fief.CalcFamilyExpenses**
- Uses: bailiff salary (bailSal), total non-bailiff salary (emplSal), total family member allowances (famAllow), family member age
- Non-home fiefs: family expenses (famExp) = bailSal / no. of fiefs in which is bailiff
- Home fief: famExp = emplSal + famAllow + (bailSal / no. of fiefs in which is bailiff)
- Family allowances (per season):
    - Age modifier (ageMod):
        - If age <=7: ageMod = 0.25
        - If age <=14: ageMod = 0. 5
        - If age <=21: ageMod = 0.75
        - If age > 21: ageMod = 1
    - Wife = £30,000 * ageMod
    - Eldest son (heir) = £40,000 * ageMod
    - Other son = £20,000 * ageMod
    - Daughter = £15,000 * ageMod
    - Other family member = £15,000 * ageMod
- Note: Either player or spouse (whoever has highest management rating) acts as ‘bailiff’ for home family expenses.

B.4 Time taken for actions (in days)

Movement:

- **Method: Form1.getTravelCost, GameClock.CalcSeasonTravMod, and Army.CalcMovementModifier**
- Uses: army size (aSize)
- Terrain modifier (terrMod):
  - Plains: terrMod = 1
  - Hills: terrMod = 1.5
  - Forrest: terrMod = 1.5
  - Mountain: terrMod = 15
  - Impassable mountain: terrMod = 91
- Army modifier (arMod):
  - % chance of army incurring movement penalty: aSize / 1000
  - arMod = 3
- Season modifier (seaMod):
  - Winter: seaMod = 2
  - Spring: seaMod = 1.5
  - Other: seaMod = 1
- Cost = (((old fief terrMod + new fief terrMod) / 2) + arMod) * seaMod

Fight a battle:

- Cost: 1

Siege:

- Non-storm round cost: 10
- Storm: 1
- Raise: 1

Pillage:

- Cost: 7-15 (random)

Examination of army:

- Cost: 1-3 (random)

Recruit army:

- Cost: 1-5 (random)

Troop exchange:

- Cost: 10-30 (random) for both drop off and pick up

Pregnancy attempt:

- Cost: 1

B.5 Victory conditions

Individual points victory, based on:

- **Method: Globals_Game.GetCurrentScores, VictoryData.CalcStatureScore, VictoryData.CalcPopulationScore, VictoryData.CalcFiefScore, VictoryData.CalcMoneyScore**
- Stature score = currentStature + (currentStature - startingStature)
- Fiefs score = (current % fiefs + (current % fiefs – starting % fiefs) / 10)
- Population score = (current % pop + (current % pop – starting % pop) / 10)

- Money score = (current % mon + (current % mon – starting % mon) / 10)

Team victory, based on historical outcome (not implemented):

- Kingdom 1 victory based on ejecting Kingdom 2 from its historical lands
- Kingdom 2 victory based on maintaining occupation of some Kingdom 2 lands
- Degree of victory depends on proportion of lands occupied

Team victory, based on total victory:

- All fiefs owned by players of one kingdom

Individual victory based on:

- Who is king after specified period of game play

6. When attributes/values are updated

At start/end of turn:

- Clock:
  - Advance season (and year if required)
- Feif:
  - Fief loyalty
  - Unrest/rebellion
  - Keep level
  - GDP
  - Fields
  - Industry
  - Treasury loss/gains
  - Population
- Army
  - Attrition due to remaining days
  - Disband if < 100 troops
- PC/NPC:
  - Ailments
  - Death by 'normal' causes
  - Childbirth
  - Death due to childbirth (mother and/or new-born)
  - Marriage
  - Movement of unemployed NPCs (1 hex)
  - If *goTo*.count > 0: resume unfinished movement
  - Extra days used up in fief = may contribute towards *Fief.bailiffDaysInFief*

Immediately:

- Fief
  - Keep level loss due to siege
  - Population loss due to pillage/siege
  - Fields level loss due to pillage/siege
  - Industry level loss due to pillage/siege
  - Treasury loss due to pillage/siege
  - Loyalty loss due to pillage/siege
- Army
  - Casualties due to combat

    - Disband due to combat
    - Movement/location (including retreat)
    - Combat/pillage
    - Attrition due to any of above
- PC/NPC
    - Health loss due to injury, possibly resulting in death
    - Stature loss/gain due to battles, sieges, pillages, marriages
    - Pregnancy of spouse
    - Engagement
    - Movement/location

APPENDIX C: OVERVIEW OF FILES AND CONTENT

The JominiEngine currently consists of the following files (listed alphabetically):

Ailment.cs: contains the Ailment class for modelling ailments affecting character health.

Army.cs: contains the Army class.

Character.cs: contains the Character, PlayerCharacter and NonPlayerCharacter classes, and associated classes for serialisation.

Fief.cs: contains the Fief class, and associated class for serialisation.

Form1.cs: contains the Form1 class used for the test client.

The following Form1 partial classes contain methods associated with specific areas of functionality, to facilitate their future removal to other game classes.

- Form1_army.cs
- Form1_battle.cs
- Form1_birth.cs
- Form1_character.cs
- Form1_CSVimport.cs
- Form1_databaseRead.cs
- Form1_databaseWrite.cs
- Form1_fief.cs
- Form1_household.cs
- Form1_journal.cs
- Form1_marriage.cs
- Form1_meetingPlace.cs
- Form1_movement.cs
- Form1_royal_overlord.cs
- Form1_siege_pillage_rebellion.cs
- Form1_sysAdmin.cs

GameClock.cs: contains the GameClock class which keep track of the game season and year.

Globals_Client.cs: contains the Globals_Client class, used to share objects required by the client.

Globals_Game.cs: contains the Globals_Game class, used to share game-specific objects required by the server. Also contains the OwnershipChallenge and VictoryData classes.

Globals_Server.cs: contains the Globals_Server class, used to share cross-game objects required by the server.

HexMapGraph.cs: contains the HexMapGraph class, used to associate fief objects with each other to form a game-world hexagon map.

Interfaces.cs: contains the interfaces used for the Observer pattern (used for game event communication).

Journal.cs: contains the Journal and JournalEvent classes, used for game event messaging.

Kingdom.cs: contains the Kingdom class, and associated class for serialisation.

Language.cs: contains the Language and BaseLanguage classes, and associated classes for serialisation.

Nationality.cs: contains the Nationality class.

Place.cs: contains the Place class, and associated class for serialisation.

Program.cs: contains the Program class, used to initiate the application.

Province.cs: contains the Province class, and associated class for serialisation.

Rank.cs: contains the Rank and Position classes, and associated class for serialisation. Also the TitleName struct for use with the Position class.

SelectionForm.cs: contains the SelectionForm class, used in the user interface for viewing and selecting game objects.

Siege.cs: contains the Siege class.

Trait.cs: contains the Trait class.

Terrain.cs: contains the Terrain class.

Utility_Methods.cs: contains the Utility_Methods class, used for sharing useful methods of general application.

APPENDIX D: TESTING

Thorough series of tests were carried out, involving both play testing and object constructor testing. Constructor tests were carried out mainly through the use of CSV import methods, but in some cases (where a CSV import method did not exist for that object type) tests were carried out by creating objects in code. The test table for the play tests are included below.

Action or circumstance	Expected outcome	Actual outcome
Fief management		
Officials, garrison, keep, or infrastructure expenditure adjusted	Expenditure is adjusted	As expected
Auto-adjust used to reduce expenditure	Expenditure is adjusted until it is below the amount in the treasury	As expected
Auto-adjust used to reduce expenditure	Expenditure is adjusted in proportion with original expenditure	As expected
Officials, garrison, keep, or infrastructure expenditure adjusted above the maximum amount permitted	Each individual amount checked and adjusted to max if necessary	As expected
Officials expenditure adjusted	Fief loyalty adjusted -10% to +10%, depending on expenditure	As expected
Officials expenditure adjusted	Fief income adjusted -10% to +10%, depending on expenditure	As expected
Garrison expenditure adjusted	Fief loyalty adjusted -10% to +10%, depending on expenditure	As expected
Keep expenditure adjusted	Keep level increased by (expenditure / 400000)	As expected
Keep expenditure set to 0	Keep level reduced by 0.15	As expected
Infrastructure expenditure adjusted	Fief fields level increased by (spend/500000)	As expected
Infrastructure expenditure adjusted	Fief industry level increased by (spend/1500000)	As expected
Infrastructure expenditure set to 0	Fief fields level decreased by 1%	As expected
Infrastructure expenditure set to 0	Fief industry level decreased by 1%	As expected
Tax rate adjusted	Fief loyalty adjusted in direct proportion to change in tax rate	As expected
Expenditure adjusted to ensure a surplus	Fief loyalty adjusted in direct proportion to size of surplus in comparison to income	As expected
Funds transferred between a fief and the home fief	Funds deducted from donating treasury and added to receiving treasury	As expected
Funds transferred between two players	Funds deducted from donating player's home treasury and added to receiving player's home treasury	As expected
Nationality barred	Nationality ID added to fief’s	As expected

	barredNationalities	
Nationality unbarred	Nationality ID removed from fief’s barredNationalities	As expected
Character barred	Character ID added to fief’s barredCharacters	As expected
Character unbarred	Character ID removed from fief’s barredCharacters	As expected
Appoint NPC as bailiff	NPC appointed	As expected
Remove NPC from bailiff	NPC removed	As expected
Appoint PC (self) as bailiff	PC (self) appointed	As expected
Set fief status to ‘U’ (unrest)	Fief income reduced to 50% when calculating surplus	As expected
Set fief status to ‘R’ (rebellion)	Fief income reduced to 0% when calculating surplus	As expected
Set fief status to ‘R’ (rebellion)	-0.1 added to owning PC's statureModifier	As expected
Change in fief status occurs	JournalEntry created and dispatched	As expected
Did not appoint a bailiff	Auto-bailiff implemented (assumes management and stature values of 3)	As expected
Ensured bailiff spent < 30 days in fief	Auto-bailiff implemented (see above)	As expected
Ensured bailiff had different language from fief	Auto-bailiff implemented when calculating effect on fief loyalty	As expected
Performed fief update	Fief population increases by 0.5%	As expected
Performed fief update	Fief GDP calculated correctly (using fields and industry revenue)	As expected
Performed fief update	Fief income calculated correctly (using GDP & tax rate, with bonuses for bailiff stats and officials expenditure)	As expected
Performed fief update	Overlord taxes calculated correctly (using income and overlord tax rate)	As expected
Performed fief update	Family expenses calculated correctly in non-home fief (bailiff wages)	As expected
Performed fief update	Family expenses calculated correctly in home fief (bailiff wages + non-bailiff wages + family allowances)	As expected
Performed fief update	Fief surplus calculated correctly: using income (modified by bailiff stats), expenses (modified by bailiff traits), family expenses (modified by bailiff stats & traits), overlord taxes	As expected
Bailiff appointed and 30 days in fief	Fief loyalty adjusted (1.25% increase in loyalty per bailiff's stature and management average above 1)	As expected
Bailiff appointed and 30 days in fief	Fief income adjusted (2.5% increase in income per management level above 1)	As expected
Bailiff appointed and 30 days in fief	Fief family expenses adjusted (2.5% decrease in family expenses per management level above 1)	As expected

Bailiff appointed, 30 days in fief and has fiefLoyalty trait	Fief loyalty adjusted according to bailiff’s trait effects (positive or negative)	As expected
Bailiff appointed, 30 days in fief and has famExpenses trait	Fief family expenses adjusted according to bailiff’s trait effects (positive or negative)	As expected
Bailiff appointed, 30 days in fief and has fiefExpenses trait	Fief expenses (not including family expenses) adjusted according to bailiff’s trait effects (positive or negative)	As expected
Fief owner has higher management rating than spouse	Fief owner’s management rating used to effect home fief family expenses	As expected
Fief owner’s spouse has higher management rating than fief owner	Spouse’s management rating used to effect home fief family expenses	As expected
Transferred fief title to NPC	Title added to NPC’s myTitles	As expected
Remove fief title from NPC	Title removed from NPC’s myTitles	As expected
Transferred ancestral fief title to NPC	Transfer not permitted	
Performed fief update	Fief tax rate adjusted as specified	As expected
Performed fief update	Fief fields level adjusted	As expected
Performed fief update	Fief industry level adjusted	As expected
Performed fief update	Fief loyalty level adjusted	As expected
Performed fief update	Fief population level adjusted	As expected
Performed fief update	Fief keep level adjusted	As expected
Performed fief update	Fief expenditures auto-adjusted if necessary	As expected
Performed fief update	Fief isPillaged reset	As expected
Performed fief update	Fief hasRecruited reset	As expected
Performed fief update	Fief bailiffDaysInFief reset	As expected
Performed fief update	Bailiff’s spare days added to bailiffDaysInFief if appropriate	As expected
Performed fief update	Fief status check performed	As expected
Performed fief update	Fief treasury adjusted	As expected
Performed fief update	Overlord taxes added to overlord’s home fief treasury	As expected
Performed fief update in fief under occupation	Overlord taxes NOT added to overlord’s home fief treasury	As expected
Performed fief update	Fief keyStatsCurrent updated	As expected
Performed fief update	Fief keyStatsPrevious updated	As expected
Performed fief update	Attrition applied to any troop detachments in fief	As expected
Performed fief update	Troop detachments in fief had days reset	As expected
Province management		
Adjusted tax rate	Tax rate is adjusted	As expected
A fief in the province is occupied by the enemy	Occupation detected	As expected
Army management		

Recruit troops	PC exits keep	As expected
Recruit troops	New army created if PC not currently leading one	As expected
Recruit troops	If PC currently leading an army, troops are added to it	As expected
Attempt to recruit troops over fief manpower limit	Troops recruited reduced accordingly	As expected
Attempt to recruit troops in fief with unrest	Troops recruited reduced by 50%	As expected
Recruit troops	Troop type ratios are in accordance with nationality recruitment ratios	As expected
Recruit troops	Days used between 1-5	As expected
Recruit troops	PC and army days adjusted	As expected
Recruit troops	Fief hasRecruited adjusted	As expected
Recruit troops in ancestral fief	Cost of each man is 500	As expected
Recruit troops in non-ancestral fief	Cost of each man is 2000	As expected
Recruit troops	Cost deducted from PC's home fief treasury	As expected
Attempt to recruit troops in fief where PC and fief languages are different, fief has loyalty >= 7	Recruitment proceeds	As expected
Attempt to recruit troops in fief where PC and fief languages are different, fief has loyalty < 7	Recruitment cancelled	As expected
Attempt to recruit troops in fief where PC is not the owner	Recruitment cancelled	As expected
Attempt to recruit troops in fief where hasRecruited = true	Recruitment cancelled	As expected
Attempt to recruit troops in fief under siege	Recruitment cancelled	As expected
Attempt to recruit troops when PC has < 1 days	Recruitment cancelled	As expected
Attempt to recruit troops when PC has less days than days taken for recruitment	Recruitment cancelled	As expected
Attempt to recruit troops when PC has less funds than are required for a single man	Recruitment cancelled	As expected
Attempt to recruit troops when PC has less funds than are required for specified number	Number requested is reduced accordingly	As expected
Attempt to recruit troops from fief in rebellion	Recruitment cancelled	As expected
Appoint PC (self) as leader	PC (self) appointed (PC.armyID and Army.leader adjusted)	As expected
Appoint NPC as leader	NPC appointed (see above)	As expected
Appoint PC/NPC as leader when are leader of another army	Warning displayed, removed from leadership of other army	As expected
Move army	Army and leader location changed	As expected

Move army	Army and leader days adjusted	As expected
Move army over 1000 in size	Travel cost adjusted if necessary	As expected
Move army	Attrition check performed	As expected
Camp army	Army and leader days adjusted	As expected
Camp army for >= 7 days	Attrition check performed	As expected
Army takes part in siege round as attacker	Attrition check performed	As expected
Army takes part in siege round as defender	Attrition check only performed if siege days > (bailiff management * 60)	As expected
Attrition check performed during winter or spring	Chance of attrition += 20	As expected
Attrition check performed during winter or spring	Attrition losses * 3	As expected
Attrition check performed	Chance of attrition reduced by average of leader's management + stature	As expected
Army has isMaintained = true	Attrition checks NOT performed	As expected
Army is maintained	Army isMaintained is adjusted	As expected
Army is maintained	Cost deducted from owning PC's home treasury	As expected
Troops transfer performed	Troops removed from donating army	As expected
Troops transfer pickup performed	Troops added to collecting army	As expected
Troops transfer pickup attempted by PC that is not donating or specified collecting PC	Pickup cancelled	As expected
PC or NPC observes non-owned army	Estimated numbers based on observer leadership (av of man/com/stat) & 'battle' skill effect	As expected
PC or NPC observes non-owned army	PC/NPC days adjusted	As expected
PC or NPC with < 1 days attempts observation of non-owned army	Observation cancelled	As expected
Army aggression level adjusted	Army aggression level updated	As expected
Army combatOdds level adjusted	Army combatOdds level updated	As expected
Army aggression level set to > 2	Army aggression level adjusted to 2	As expected
Army aggression level set to < 0	Adjustment cancelled	As expected
Army combatOdds set to < 0	Adjustment cancelled	As expected
Enemy army attacks an army with aggression = 0 which is outside keep	Army attempts to retreat	As expected
Army with aggression = 0 enters keep	Army aggression level adjusted to 1	As expected
Enemy army attempts siege or pillage in fief where an army with aggression = 0 which is	Siege or pillage permitted	As expected

outside keep		
Enemy army attempts siege or pillage in fief where an army with aggression = 1 which is outside keep	Siege or pillage permitted	As expected
Enemy army attacks an army with aggression = 1 which is outside keep	Army will defend if battle odds <= combatOdds, otherwise attempts to retreat	As expected
Enemy army performs siege round in fief with an army with aggression = 1 which is inside keep	Army will sally and attack if battle odds >= combatOdds	As expected
Enemy army attempts siege or pillage in fief where an army with aggression = 2 which is outside keep	Siege or pillage cancelled	As expected
Enemy army attacks an army with aggression = 2 which is outside keep	Army will defend	As expected
Enemy army performs siege round in fief with an army with aggression = 2 which is inside keep	Army will sally and attack if battle odds >= combatOdds	As expected
Seasonal update performed	Attrition check performed for every 7 army days remaining (unless maintained)	As expected
Seasonal update performed	Army disbanded if has < 100 troops	As expected
Seasonal update performed	Army and leader days reset	As expected
Seasonal update performed	Army isMaintained reset	As expected
Battle		
Battle occurs	JournalEntry created and dispatched	As expected
Battle occurs	Chance of bringing defending army to battle correctly calculated (using appropriate battleProbabilities)	As expected
Battle occurs	Army leadership value (LV) correctly calculated	As expected
Battle occurs	Leader trait effects correctly applied to LV	As expected
Battle occurs	Army combat value correctly calculated	As expected
Battle occurs	Character combat values correctly calculated	As expected
Battle occurs	Army battle value modifier correctly calculated	As expected
Battle occurs	Battle odds correctly calculated	As expected
Battle occurs	Calculation of victory correctly calculated	As expected
Battle occurs	Chance of victory correctly calculated	As expected
Battle occurs	Casualties correctly calculated	As expected
Battle occurs and casualties are	Casualties removed from army troop	As expected

incurred	numbers	
Battle occurs	Chance of retreat correctly calculated, taking into account casualties and standing orders	As expected
Retreat occurs	Retreat number of hexes 1-2	As expected
Retreat occurs	Army will attempt to retreat into friendly fiefs where possible	As expected
Battle occurs	Army owners' statureModifier adjusted accordingly	As expected
Battle occurs	Army and leader days adjusted	As expected
Army disbanded	Army removed from fief (armiesInFief), owner (myArmies), leader (armyID) and siege	As expected
Besieging army disbanded	Siege ended	As expected
Battle occurs	Character chance of injury correctly calculated (including effect of character health and combat rating)	As expected
Injury occurs	JournalEntry created and dispatched	As expected
Injury occurs	If ailment effect > 4, ailment minimumEffect set to 1	As expected
Siege, pillage and rebellion		
Siege starts	JournalEntry created and dispatched	As expected
Siege starts	Garrison size correctly calculated (based on garrison expenditure and fief population)	As expected
Siege starts	Garrison troop types correctly apportioned (based on nationality recruit ratios)	As expected
Siege storm round occurs	Army battle values correctly calculated, including effect of keep level	As expected
Siege storm round occurs	Chance of success correctly calculated	As expected
Siege negotiation round occurs	Chance of success correctly calculated	As expected
Siege round occurs	JournalEntry created and dispatched	As expected
Siege round occurs	Casualties correctly calculated and army troop numbers adjusted	As expected
Siege round occurs	Keep level reduction correctly calculated, fief keep level adjusted	As expected
Siege storm round successful	Fief pillage occurs	As expected
Siege storm round successful	Captives are taken (only defending PC, his NPCs and other enemy PCs)	As expected
Captives taken	Ransom correctly calculated (% of PC GDP, non-family NPC wage, family NPC allowance)	As expected
Captives taken	Ransom removed from captive's home treasury and added to victor's home treasury	As expected
Siege successful	Fief ownership transferred	As expected
Siege successful	Fief title transferred	As expected
Siege successful	Fief loyalty altered by +10%/-10% if	As expected

	new/old owner is ancestral owner	
Siege successful	Fief bailiff removed and bailiffDaysInFief reset	As expected
Siege successful	Fief status check performed	As expected
Siege successful	New owner, his NPCs and nationality unbarred	As expected
Siege storm round successful	Victor's statureModifier adjusted by (0.1 * population/10000)	As expected
Siege storm round fails	Victor's statureModifier adjusted by (-0.1 * population/10000)	As expected
Siege negotiation round successful	Victor's statureModifier adjusted by (0.2 * population/10000)	As expected
Siege successful	Siege ended	As expected
Siege ended	JournalEntry created and dispatched	As expected
Siege ended	Defending garrison disbanded	As expected
Siege ongoing	All fief expenditure except garrison reduced to 0	As expected
Siege ongoing	Fief income reduced to 0	As expected
Siege ongoing	Fief family expenses (except bailiff salary) reduced by 50%	As expected
Seasonal update performed	If besieging army disbanded, siege ended	As expected
Seasonal update performed	Days of all siege objects reset and synchronised	As expected
Pillage occurs	JournalEntry created and dispatched	As expected
Pillage occurs	Pillage multiplier correctly calculated (based on number troops per 1000 population)	As expected
Pillage occurs	Fief population adjusted	As expected
Pillage occurs	Fief loyalty adjusted	As expected
Pillage occurs	Fief fields adjusted	As expected
Pillage occurs	Fief industry adjusted	As expected
Pillage occurs	Fief treasury adjusted	As expected
Pillage occurs	Fief isPillaged adjusted	As expected
Pillage occurs	Pillaging army and leader days adjusted	As expected
Pillage occurs	Amount of money pillaged correctly calculated (based on fief GDP, number of days taken, jackpot)	As expected
Pillage occurs	Proportion of money taken by pillaging army owner correctly calculated (based on owner's stature)	As expected
Pillage occurs	Proportion of money taken by pillaging army transferred to his home treasury	As expected
Pillage occurs in fief that speaks same BaseLanguage or language as pillaging army owner	Pillaging army owner's stature adjusted negatively	As expected
Quelling of rebellion attempted	Chance of success correctly calculated	As expected

	(taking into account quelling army leader's LV and quelling army owner's ancestral ownership)	
Quelling of rebellion failed	Quelling army retreats 1 hex	As expected
Quelling of rebellion failed	JournalEntry created and dispatched	As expected
Quelling of rebellion succeeds	Quelling army owner assumes ownership of fief	As expected
Quelling of rebellion succeeds	Fief status set to 'C'	As expected
Quelling of rebellion succeeds	JournalEntry created and dispatched	As expected
Household		
Household information is displayed	NPC responsibilities correctly derived: shows combination of bailiff duties (multiple), army duties or 'unspecified'	As expected
Household information is displayed	NPC function correctly derived: shows 'employee', specific family role or 'family member'	As expected
Household: Marriage		
Proposal is made	Correct conditional checks performed (bride & groom age >= 14, bride = female, groom = male, bride & groom not married or engaged, bride daughter of played PC, groom son of played PC or played PC themselves, bride & groom not in same family)	As expected
Proposal replied to	Original proposal message amended to show reply	As expected
Proposal accepted	JournalEntry created and dispatched	As expected
Proposal accepted	Marriage added to scheduled events	As expected
Proposal accepted	Bride & groom fiancée adjusted	As expected
Proposal accepted	Marriage NOT added to scheduled events	As expected
Season update performed when bride & groom separated by siege	Marriage postponed	As expected
Bride or groom die during season update	Marriage cancelled	As expected
Head of family dies during season update and is no heir	Marriage cancelled	As expected
Marriage cancelled	JournalEntry created and dispatched	As expected
Season update performed	Marriage processed	As expected
Marriage processed	JournalEntry created and dispatched	As expected
Marriage processed	Bride & groom fiancée adjusted	As expected
Marriage processed	Bride & groom spouse adjusted	As expected
Marriage processed	Bride familyID adjusted	As expected
Marriage processed	Bride added to new head of family's myNPCs and removed from old head of family's myNPCs	As expected
Marriage processed	Bride's location adjusted	As expected
Marriage processed and bride's	New head of family's statureModifier	As expected

family is of higher rank than groom's	adjusted (0.4 per difference in rank)	
Household: Pregnancy and childbirth		
Pregnancy attempted	Correct conditional checks performed (husband and wife spouse = each other, spouse in same fief, spouse not separated by siege, spouse not pregnant, husband and wife have enough days)	As expected
Pregnancy attempted	Husband and wife inKeep synchronised, and days adjusted	As expected
Pregnancy attempted	Chance of success correctly calculated (taking into account wife age and husband and wife virility)	As expected
Pregnancy success	JournalEntry created and dispatched	As expected
Birth processed	JournalEntry created and dispatched	As expected
Birth processed	New NPC successfully generated. [JournalEntry created and dispatched]	As expected
Birth processed	New NPC's attributes based on that of parents (maxHealth, virility, management, combat, traits)	As expected
Birth processed	Mother and baby undergo death check, using specific childbirth conditions	As expected
Household: Inheritance		
Death occurs (PC with heir)	PC's heir is promoted to PC status and inherits PC's properties and position	As expected
NPC inherits	Deceased's titles transferred to heir	
NPC inherits	Deceased's owned fiefs transferred to heir	
NPC inherits	Deceased's owned provinces transferred to heir	
NPC inherits	Deceased's myNPCs transferred to heir	
NPC inherits	Heir replaces his old NPC in fief	
NPC inherits	Heir's myNPCs all have familyID changed to heir's ID	
NPC inherits	Deceased's ancestral ownerships transferred to heir	
NPC inherits	Deceased's armies and sieges transferred to heir	
NPC inherits	Deceased's OwnershipChallenges transferred to heir	
NPC inherits	Deceased's VictoryData entry amended to show heir's ID	
Death occurs (PC without heir)	King inherits PC's properties and position	As expected
king inherits	Deceased's armies are disbanded and sieges ended	
king inherits	Deceased's NPC's inEntourage set to	

	false	
king inherits	Deceased's non-family NPC's transferred to king	
king inherits	Deceased's family NPC's cast out (salary set to 0, familyID set to null, inKeep set to false, titles transferred to king, marriages and births cancelled)	
king inherits	Deceased's OwnershipChallenges deleted	
king inherits	Deceased's VictoryData entry removed	
Household: Hiring and firing		
Prospective employee details displayed	NPC potential salary correctly calculated (taking into account fief management rating, leadership rating, current salary, hiring PC stature, current employer stature)	As expected
Employment offer made	Chance of success correctly calculated (taking into account hiring PC's npcHire traits)	As expected
Employment offer made	An offer > 10% higher than potential salary will always be accepted	As expected
Employment offer made	An offer > 10% lower than potential salary will always be rejected	As expected
Employment offer made	An offer lower than the previous offer will always be rejected	As expected
NPC hired	NPC added to employer's myNPCs	As expected
NPC hired	NPC's employer attribute updated	As expected
NPC hired	NPC's salary attribute updated	As expected
NPC hired	NPC's lastOffer attribute cleared	As expected
NPC hired	NPC removed from previous employer	As expected
NPC fired	NPC removed from employer's myNPCs	As expected
NPC fired	NPC's employer attribute set to null	As expected
NPC fired	NPC's salary attribute set to null	As expected
NPC fired	NPC removed from bailiff positions	As expected
NPC fired	NPC removed from army leader positions	As expected
NPC fired	NPC's inEntourage attribute set to false	As expected
NPC fired	NPC's titles transferred to employer	As expected
NPC fired	NPC's goTo attribute cleared	As expected
Character		
Character performs an action that can be affected by their traits	Trait effect is correctly applied (battle, famExpense, fiefLoyalty, fiefExpense, siege, time, death, npcHire)	As expected
Character involved in action	Base stature correctly calculated	As expected

that requires use of stature	(based on highest rank, sex and age)	
Character involved in action that requires use of stature	Base stature correctly modified by character’s statureModifier	As expected
Character involved in action that requires use of health	Base health correctly calculated (based on maxHealth and age)	As expected
Character involved in action that requires use of health	Base health correctly modified by character’s ailments	As expected
Season update performed	Character’s days reset	As expected
Season update performed	Character’s ailments processed (removing any that have effect = 0)	As expected
Season update performed	Unemployed non-family NPCs moved 1 hex (randomly chosen)	As expected
Season update performed	Death check performed	As expected
Death check performed	Chance of death correctly calculated (based on death multiplier and health, influenced by ‘death trait)	As expected
Death occurs (character)	JournalEntry created and dispatched if appropriate	As expected
Death occurs (character)	Character isAlive set to false	As expected
Death occurs (character)	Character removed from fief’s charactersInFief	As expected
Death occurs (character)	Character removed from spouse’s spouse attribute	As expected
Death occurs (character)	Character removed from bailiff and leader positions	As expected
Death occurs (character)	Character childbirth and marriage events removed from scheduled events (and is removed from fiancee’s fiancée attribute)	As expected
Death occurs (NPC)	NPC removed from head of family or employer’s myNPCs	As expected
Death occurs (NPC)	NPC’s titles returned to owner	As expected
Death occurs (non-family NPC)	NPC re-spawned in random fief of same language	As expected
Death occurs (PC)	PC removed as holder of any positions	As expected
Movement		
Character moves single hex	Movement cost correctly calculated: (source fief cost + target fief cost) / 2 (amended by season and army size modifiers)	As expected
Character moves single hex	Character removed from old fief’s charactersInFief and added to new fief,s charactersInFief	As expected
Character moves single hex	Character’s location attribute updated	As expected
Character moves using ‘take exact route’ method	Character moves to specified fiefs in correct order	As expected
Character moves using ‘take exact route’ method without	Movement is halted and resumes after the season update	As expected

sufficient days to finish move		
Character attempts to use 'take exact route' method but incorrect direction entry is entered	Movement instructions curtailed at incorrect entry; movement performed	As expected
Character moves using 'move to' method	Movement performed by taking route of least cost	As expected
Character moves using 'move to' method without sufficient days to finish move	Movement is halted and resumes after the season update	As expected
Character attempts to use 'move to' method but incorrect fief ID is entered	Movement cancelled	As expected
Character uses camp to remain in place	Camped days are counted towards fief bailiffDaysInFief if appropriate	As expected
Character uses camp to remain in place when leading a besieging army	The days of all siege objects are synchronised with the character	As expected
Character attempts to enter keep from which he has been barred	Keep entry cancelled	As expected
Character attempts to enter keep from which his nationality has been barred	Keep entry cancelled	As expected
Character leading a friendly army attempts to enter keep in containing another friendly army	Keep entry cancelled	As expected
Character leading a non-friendly army attempts to enter keep	Keep entry cancelled	As expected
PC attempts to enter keep from which the nationality of one of his entourage has been barred	PC and entourage have inKeep set to true (PC 'vouches' for his entourage)	As expected
PC attempts to enter keep from which one of his entourage has been barred	PC can enter but not the member of the entourage who has been banned	As expected
PC exits keep	PC and entourage all have inKeep set to false	As expected
Title and positions		
Fief owner grants title to NPC	Title removed from PC's myTitles and added to NPC's myTitles. [JournalEntry created and dispatched]	As expected
Fief owner grants title to NPC	Fief's titleholder attribute amended	As expected
Fief owner attempts to grant title of highest ranking fief to NPC	Transfer cancelled	As expected
Fief owner attempts to grant title of fief for which he is ancestral owner to NPC	Transfer cancelled	As expected
King attempts to grant title of	Transfer processed normally	As expected

fief for which he is ancestral owner to NPC		
King grants province title to played PC	Title removed from king's myTitles and added to PC's myTitles. [JournalEntry created and dispatched]	As expected
King grants province title to played PC	Province's titleholder attribute amended	As expected
King grants province title already held by another played PC	Title removed from other PC's myTitles	As expected
King grants position to played PC	Province's officeholder attribute amended. [JournalEntry created and dispatched]	As expected
King grants position to played PC	PC's statureModifier adjusted accordingly	As expected
King grants position already held by another played PC	Old holder's statureModifier adjusted accordingly	As expected
Ownership challenges		
PC lodges ownership challenge for province or kingdom	New OwnershipChallenge created and added to ownershipChallenges. [JournalEntry created and dispatched]	As expected
PC attempts to lodge ownership challenge for province he owns	Challenge cancelled	As expected
PC attempts to lodge ownership challenge for province that is already the subject of a challenge	Challenge cancelled	As expected
Season update preformed	Ownership challenges processed	As expected
Ownership challenge processed and challenger satisfies criteria	Challenge counter incremented	As expected
Ownership challenge processed and challenger does not satisfy criteria	Challenge removed from ownershipChallenges. [JournalEntry created and dispatched]	As expected
Ownership challenge processed and counter reaches 4	Ownership of place transferred from old owner to new owner, including title. [JournalEntry created and dispatched]	As expected
Successful challenge for ownership of kingdom	King attribute of old owner transferred to new owner . [JournalEntry created and dispatched]	As expected

Table 12: Tests performed during play testing

CSV import specification

✓ Test tables?

List of containers used in UI (Form1 and SelectionForm)